Table of Contents

首页

> 本文档由 **Docsify** 生成，目前处于文本修缮的阶段，修缮进度可在 **Trello** 上查看，囿于个人业余时间精力有限，如果有兴趣加快本文档建设的朋友可以从右上角的 **Github** 链接处查看协作方式。

本文档并非初学者教程，并不会教你如何编程。尽管如此，每一主题都被粗糙地设计为顺序阅读，其中会穿插一些如 “先前看过的” 或是 “尚未了解的” 引用链接。在一些情况下，一个较为靠前的章节会为了便于解释相关主题而使用一些后续章节才详细展开的内容，这些内容都会附上相应的跳转链接，提前阅读这些主题通常来说问题不大。

我们会使用很多 Haxe 源代码来保持理论与实践之间的桥接。这些代码示例通常是完整的程序，并带有一个 main 函数，它们都可以被编译。不过也有时候只有最关键的部分被展示出来。源代码看起来就像这样：

```
Haxe code here
```

偶尔我们会演示 Haxe 代码是如何被生成的，对于这些内容，通常会以 JavaScript 目标平台的输出用作解释。此外，在本文档中我们定义了一组术语。主要用于引入一个新的类型或是一个 Haxe 专用术语。我们不会定义每个新引入的概念，譬如什么是类，以避免弄乱文本。定义看起来像这样：

> 定义: 定义名称 定义描述

文档中的一些地方还有一些花絮框。其中包含了一些题外信息，如为什么在 Haxe 的开发中做出某些决定，或者过去的 Haxe 版本中的某些实际功能的改变等。这类信息通常不是很重要，你可以选择跳过，因为它们只是为了传达一些花絮。花絮看起来像这样：

> 花絮: 关于花絮 花絮内容

1.Haxe介绍

本章内容：

1.1.Haxe是什么

Haxe 由一个高级的、开源编程语言和一个编译器构成。它允许使用一个 ECMAScript 规范面向对象的语法编写编译程序到多种目标平台。得益于适当的抽象，它可以维护一个单一基础代码，并支持编译到多种目标平台。

Haxe 是强类型，但是类型系统可以在需要的地方被推翻。利用类型信息，Haxe 类型系统可以在编译时检测那些可能只在目标语言运行时才被注意到的错误。此外，类型信息可以被目标语言生成器用来生成优化的、健壮的代码。

目前, Haxe 支持九个目标语言用于不同的用例：

名称	输出类型	主要用途
JavaScript	Sourcecode	Browser, Desktop, Mobile, Server
Neko	Bytecode	Desktop, Server
PHP	Sourcecode	Server
Python	Sourcecode	Desktop, Server
C++	Sourcecode	Desktop, Mobile, Server
ActionScript 3	Sourcecode	Browser, Desktop, Mobile
Flash	Bytecode	Browser, Desktop, Mobile
Java	Sourcecode	Desktop, Server
C#	Sourcecode	Desktop, Mobile, Server

第一部分余下的内容简要概述了 一个Haxe 程序的模样，和自2005年开始以来的进化。

类型（第2章）介绍了 Haxe 中七种不同的类型，和它们之间的交互。在 类型系统（第3章） 中将继续对类型的讨论，如 类型统一、类型参数和类型推断等。

类字段（第4章） 所有内容都是关于 Haxe 类的结构，其中包括一些其它的话题、属性的处理、内联字段和泛型函数等。

在 表达式（第5章） 中我们了解如何通过使用表达式实际上上程序做些什么。

语言特性（第6章） 详细描述了一些 Haxe 的特性，如模式匹配、字符串插值 和无用代码消除。这总结了 Haxe 语言参考。

接下来是 Haxe 编译器参考，首先在进入 编译器的高级功能（第8章） 之前讲述基础的 编译器用法（第7章）。然后，将探索令人兴奋的内容， Haxe 的 宏（第9章），了解一些常见的如何被大大简化。

后面的章节中， 标准库（第10章） 探索 Haxe 标准库中重要的类型和概念。然后学习 Haxe 的包管理系统 Haxelib（第11章）。

Haxe 抽象了许多目标平台的差异，但是有时和目标平台直接交互也是非常重要的，在 目标平台细节（第12章） 中，我们将讨论这个主题。

1.2.关于本文档

本文档翻译自 Haxe 3 官方手册。因此，它不是一个初学者教程，并不会教你如何编程。然而，主题都被粗略设计为便于阅读，对相关话题进行引用，如，一些之前看到的，和尚未了解的。在一些情况下，如果可以简化解释，一个较早的章节会使用后面的一些章节的信息。这些引用将有相应的连接，提前阅读这些话题，通常不是什么问题。

我们会使用很多 Haxe 源代码来保持理论和实际的连接。这些代码示例通常是完整的程序，并带有一个 `main` 函数，它们都可以被编译。然而，有时候只有最重要的部分被展示出来。源代码看起来就像这样：

```
Haxe code here
```

偶尔地，我们会演示 Haxe 代码怎样被生成，对于这些，通常会显示 JavaScript 目标平台。此外，在本文档中我们定义了一组术语。主要用于引入一个新的类型或者一个 Haxe 专用术语。我们不会定义每个新引入的概念，譬如什么是类，以避免弄乱文本。定义看起来像这样：

> 定义: 定义名称 定义描述

文档中的一些地方还有一些花絮框。这些包括非正式的信息，如为什么在 Haxe 的开发中做出某些决定，或者过去的 Haxe 版本中的某些实际功能的改变等。这类信息通常不是很重要，你可以选择跳过，因为它们只是为了传达一些花絮。花絮看起来像这样：

> 花絮: 关于花絮 花絮内容

1.2.1.作者及贡献者

本文档大部分内容是由 Simon Krajewski 就职于 Haxe 基金会时 编写。我们感谢这些人的贡献：

- Dan Korostelev：追加内容和编辑
- Caleb Harper：追加内容和编辑
- Josefiene Pertosa：编辑
- Miha Lunar：编辑
- Nicolas Cannasse：Haxe 之父

1.2.2.License

1.3Hello World

下面的程序被编译并运行后，将打印“ `Hello World` ”：

```
class Main
{
    static public function main():Void {
        trace("Hello World");
    }
}
```

可以保存上面的代码到一个 `Main.hx` 文件，并使用 Haxe 编译器运行命令： `-main Main --interp` 。然后生成如下输出： `Main.hx:3: Hello world` 。从其中我们可以了解到以下几件事情：

- Haxe 程序被保存为 `.hx` 扩展名的文件
- Haxe编译器是一个命令行工具，可以在调用时使用参数
- 如 `-main Main` 和 `--interp`
- Haxe 程序使用类（Main，首字母大写），类中包含函数（main，首字母小写）
- 包含主要 Haxe 类文件的名字和类的名字相同（本例中的 Main.hx ）。

1.4.Haxe的历史

Haxe 项目开始于2005年8月22日，由法国开发者 Nicolas Cannasse 创建，作为流行的开源 ActionScript 2编译器 MTASC（Motion-Twin Action Script Compiler）和内部的MTypes语言继任者，尝试将类型推断的应用成为一个面向对象语言。Nicolas 长期的编程语言设计热情，和混合不同技术作为他使用 Motion-Twin 进行游戏开发工作中一部分的新机会出现，催生了一个全新语言的产生。

开始时 Haxe 被拼写为 haXe，之后，它的 beta 版本在2006年2月发布，附带了第一个支持的目标平台 AVM2（Adobe Virtual Machine ）字节码， 和 Nicolas 自己的 Neko 虚拟机 3 。

Nicolas Cannasse，目前仍是 Haxe 项目的领导者，一直致力于使 Haxe 更加智能，随后在2006年5月发布了 Haxe 1.0 。第一个主要版本开始支持 JavaScript 代码生成，并已经有一些定义今天的 Haxe 的基础，如类型推断和结构子类。

Haxe 1 在两年中经历了几个小的版本，在2006年8月添加了支持 haxelib工具的 Flash AVM 2 目标平台，在2007年3月增加的对 ActionScript 3 目标的支持。这几个月里，Haxe 更加聚焦于稳定性的提升，它的几个小版本里的 bug 得到修复。

Haxe 2.0 在2008年7月发布，引入了 PHP 目标平台，这要感谢Franco Ponticelli 。在 Hugh Sanderson 的努力下，在2007年7月的 Haxe 2.04 版本中加入了 C++ 目标平台。

正如Haxe 1 那样，接下来的几个月，版本继续保持稳定。在2011年1月，Haxe 2.04 发布，开始支持 宏。几乎同时，Bruno Garcia 加入团队，作为 JavaScript 平台的维护者，并在随后的 2.08 和 2.09 版本中做了更大的改进。

在 2.09版本之后，Simon Krajewski 加入团队，开始了 Haxe 3 的开发。此外，Java 和 C# 目标平台也找到加入到 Haxe 版本中的方式。然后，决定建立最终的 Haxe 2 版本，即2012年7月发布的 Haxe 2.10 。

2012年底，Haxe 3 项目开始， Haxe 编译器团队，现在由最近建立的 Haxe 基金会提供资金支持，开始聚焦于下一个主要版本。Haxe 3 随后发布于2013年5月。

2.类型

Haxe 编译器利用丰富的类型系统，可以在编译时检测程序中类型相关的错误。类型错误是指对一个给定类型执行了一个无效操作，比如，除以一个字符串、尝试访问一个整数的字段，或者调用一个函数而没有传递足够（或太多）的参数。在一些语言中，这些额外的安全性考虑通常需要程序员消耗额外的精力，因为程序员被要求使用显式分配类型的语法结构：

> *The Haxe Compiler employs a rich type system which helps detect type-related errors in a program at compile-time. A type error is an invalid operation on a given type such as dividing by a String, trying to access a field of an Integer or calling a function with too few (or too many) arguments.*
>
> *In some languages this additional safety comes at a price because programmers are forced to explicitly assign types to syntactic constructs:*

```
var myButton:MySpecialButton = new MySpecialButton(); // As3
MySpecialButton* myButton = new MySpecialButton(); // C++
```

显式地类型声明在 Haxe 中并不是必须的，因为编译器可以推断类型：

> *Explicit type annotations are not required in Haxe, because the compiler can **infer** the type:*

```
var myButton = new MySpecialButton(); // Haxe
```

我们会在 类型推断（第3.6节） 中对类型推导的细节进行说明。就目前而言，我们可以肯定的说上面代码中的变量 `myButton` 是 `MySpecialButton` 类的一个实例。

> *We will explore type inference in detail later in Type Inference. For now, it is sufficient to say that the variable `myButton` in the above code is known to be an **instance of class** `MySpecialButton`.*

Haxe 的类型系统能识别七个类型组（Type Group）：

- 类实例：某个给定的类或者接口的对象
- 枚举实例：Haxe 枚举类型中的一个值
- 结构类型：一个匿名结构，即，一个被命名的字段的集合
- 函数类型：一个由一些参数和一个返回值组成的复合类型
- 动态类型：一个万能类型，可以和任何类型兼容
- 抽象类型：一个编译时类型，会在运行时被一另外一种类型表示
- 单形类型：一个 未知的（unknown） 类型，会在之后转变为一个确定的类型

> *The Haxe type system knows seven type groups:*
>
> - *Class instance: an object of a given class or interface*
> - *Enum instance: a value of a Haxe enumeration*
> - *Structure: an anonymous structure, i.e. a collection of named fields*
> - *Function: a compound type of several arguments and one return*
> - *Dynamic: a wildcard type which is compatible with any other type*
> - *Abstract: a compile-time type which is represented by a different type at runtime*
> - *Monomorph: an unknown type which may later become a different type*

下一章中，我们将描述这些类型组中的每个类型，以及它们之间的联系。

> 定义 复合类型： 复合类型是指含有子类型的类型。这其中包括使用了 类型参数（第3.2节） 的任何类型以及 函数类型（第2.6节）。

2.1.基本类型

基本类型包括 `Bool`， `Float` 和 `Int`。很容易在语法中通过值认出它们：

- `Bool` 类型的值 `true` 和 `false`
- `Int` 类型的值 `1`，`0`，`-1` 和 `0xFF0000`
- `Float` 类型的值 `1.0`，`0.0`，`-1.0`，`1e10`

在 Haxe 中，基本类型并不是 类（第2.3节）。它们被实现为 抽象类型（第2.8节），并且与编译器内部的操作符处理进行绑定，将在后面的小节中展开讲述。

2.1.1.数值类型

> 定义： Float 表示一个双精度 IEEE 64 位浮点数
>
> 定义： Int 表示一个整数

尽管可以在任何期待 `Float` 类型的位置放置一个 `Int` 类型（也就是说， `Int` 可以被赋值到 `Float` 类型上，或者说 `Int` 类型可以统一为 `Float` 类型），但是反之则不行：把一个 `Float` 类型的值赋值给 `Int` 类型可能会导致精度丢失，Haxe 不允许此类隐式转换。

补充：

你可以使用 Haxe 标准库的函数把 `Int` 类型转换为 `Float` 类型，比如：

```
var intDivision = Std.int(6.2/4.7)
```

2.1.2.溢出

出于性能原因，Haxe 编译器不实施任何溢出行为。溢出检查的任务落到目标平台。这里是一些溢出行为的平台特定提示：

- C++，Java，C#，Neko，Flash：32位带符号整数有通常的溢出惯例。
- PHP，JS，Flash 8：没有原生的 Int 类型，如果数值到达最大浮点数限制（2^52）则会发生精度的损失。

另外， `haxe.Int32` 和 `haxe.Int64` 类可以用来确保正确的溢出行为，与平台无关，而额外的计算成本取决于目标平台。

2.1.3.Bool类型

> 定义 Bool： Bool 类型表示一个非 `true` 即 `false` 的值。

Bool 类型的值经常出现在条件中，例如 `if(5.16)` 和 `while(5.14)` 。下面的运算符接受并返回 Bool 类型的值：

- && （和）
- || （或）
- ！（非）

Haxe 保证混合的布尔表达式在运行时从左到右被评估，并且只进行需要的评估。 例如，表达式 `A && B` 会首先执行 `A` ，然后再执行 `B` ，并且只有在 `A` 的结果为 `true` 的时候才继续执行 `B` 。同样，表达式 `A || B` 在 `A` 的执行结果为 `true` 的时候不会执行 `B` ，因为这时 `B` 的值是无关紧要的。在如下情况这很重要：

```
if (object != null && object.field == 1) { }
```

如果 `object` 为 `null` 的时候访问 `object.field` 会导致一个运行时错误，但是对 `object != null` 的检查则会防止它。

2.1.4.Void类型

> 定义 Void： `Void` 表示一个类型的缺失。它用来表达一些东西（通常是一个函数）没有值。

`Void` 是类型系统中的一个特殊情况，因为它实际上不是一个类型。它用来表达一个类型的缺省，多数时候用于函数参数和返回类型。我们在开始的 “Hello World” 例子中已经见到过 `Void` 。

```
class Main {
    static public function main():Void {
        trace("Hello World");
    }
}
```

在 函数类型（第2.6节） 中将会探索函数类型的详细信息，这里快速预览一下：例子中函数 `main` 的类型为 `Void->Void` ，表示没有参数和返回值。Haxe 不允许 `Void` 类型的字段和变量，如果尝试这样声明，它会发出抱怨：

```
// Void类型的参数和变量是不被允许的
var x:Void;
```

注：【环境：**Haxe 3.3.0-rc.1**】在实际编写测试代码的时候，尝试在类的 `main` 函数内定义 `var x:Void;` 并未报错，而作为类字段进行定义则不能编译，提示 **“Fields of type Void are not allowed”**。

2.1.5.为空性

> 定义 可空的： 在 Haxe 中如果 `null` 对于一个类型是有效的值，则认为类型是可空的。

在编程语言中有一个单一、清楚的为空性定义非常常见。然而，Haxe 由于其平台无关的性质，需要在这个问题中找到一个妥协：在某些目标语言中允许且事实上默认给任意类型赋 `null` 值，而另外一些语言则甚至不允许 `null` 值作为某种类型的值。因此要明确区分出两种类型的语言：

> 定义 静态目标 静态目标平台的类型系统不允许 `null` 值作为基本类型的值。比如 Flash，C++，Java 和 C# 目标平台。
>
> 定义 动态目标： 动态目标平台的类型使用更为宽松，并允许基本类型使用 `null` 值。比如 JavaScript，PHP，Neko 和 Flash 6-8 目标平台。

如果编译至动态目标下使用 `null` 时不需要担心；然而，如果要编译至静态目标平台可能需要一些考虑。首先，基本类型的值会被初始化为它们的默认值。

> 定义：默认值 静态目标语言中，基本类型的默认值如下：
>
> - `Int` : `0`

> - `Float`：Flash 平台下初始值为 `NaN`，其他平台下为 `0.0`
> - `Bool`：`false`

因此，静态目标平台下 Haxe 编译器不允许给基本数据类型赋 `null` 值，如果要这么做，基本数据类型需要包装为 `Null<T>` 类型。

```
// 静态目标平台报错
var a:Int = null;
// 允许的
var b:Null<Int> = null;
```

同样，基本类型不能跟 `null` 进行比较，除非它被包装过：

```
var a : Int = 0;

// 静态目标平台报错
if( a == null ) { ... }

var b : Null<Int> = 0;
if( b != null ) { ... } // 允许的
```

这一限制可延伸至所有 一致性检查（第3.5节） 被执行的情况。

> 定义 `Null<T>`： 在静态目标平台下，可以使用 `Null<Int>`、`Null<Float>`、和 `Null<Bool>` 类型来允许 `null` 作为一个有效值。在动态目标平台下，这样做没有影响。`Null<T>` 也可以被用于其它类型以表示 `null` 是一个允许的值。

如果一个 `null` 值被“隐藏”在 `Null<T>` 或者 `Dynamic` 中，并被赋值给一个基本类型，会对其赋默认值：

```
var n : Null<Int> = null;
var a : Int = n;
trace(n); // 输出 null
trace(a); // 在静态目标平台输出 0
```

2.2.6.可选参数和为空性

可选参数在考虑为空性时需要注意，需要在不可空的原生可选参数和可能为空的 Haxe特定（Haxe-Specific) 可选参数之间进行区分。这种区分是通过使用问号可选参数来实现的。

```
// x是一个原生 Int（不能为空）
function foo(x : Int = 0) {}

// y是一个 Null<T>类型 (可空)
function bar( ?y : Int) {}

// z同样是 Null<T>
function opt( ?z : Int = -1) {}
```

> 花絮：Argument 和 Parameter 在一些其它的编程语言中，argument 和 parameter 可以交替使用。在 Haxe 中，argument 在针对方法的时候使用，而 parameter 是指 类型参数（第3.2节）。

2.3.类实例

和许多面向对象语言类似，类是大多数 Haxe 程序中主要的数据结构。每个 Haxe 类都有一个确定的名字，一个隐含的路径和零或者多个类字段。这里我们将关注类的一般结构及它们之间的关系，而 类字段（第4章） 的详细内容将在之后展开。

以下代码示例作为本节剩余部分的基础：

```
class Point {
    var x : Int;
    var y : Int;
    public function new(x,y) {
        this.x = x;
        this.y = y;
    }
    public function toString() {
        return "Point("+x+","+y+")";
    }
}
```

从语义上讲，这个类表示二维空间内的一个点，但是这里它是什么并不重要。我们来描述一下这个结构：

- 关键字 `class` 表示我们定义一个类
- `Point` 是类的名称，可以使用任何符合类型标识符规则的字符
- 包围在花括号 `{}` 中间的是类的字段
- 它由两个 Int 类型变量字段 `x` 和 `y` 组成
- 后面是一个特定的函数字段叫做 `new` ，它是类的构造函数
- 还有一个普通的函数 `toString`

在 Haxe 中有一个特殊类型，可以兼容所有的类：

> 类型：Class 这个类型可以兼容所有类型，也就是说，所有类（而不是它们的实例）可以被分配给它。在编译时，`Class` 是所有类的基础类型。然而，这个关系并不会反映在生成的代码中。当一个 API 需要的一个值是一个类而非某个特定的类型时，可以使用这个类型。这应用到 Haxe 反射API （第10.7节） 中的一些方法。

2.3.1.类的构造函数

类的实例通过调用类的构造函数（一个通常称为实例化的过程）创建。类实例的另一个称呼叫做对象。然而，我们更倾向于使用术语“类的实例”来强调 类/类实例 及 枚举/枚举实例（第2.4节）之间的类比。

```
var p = new Point(-1, 65);
```

这会生成一个 `Point` 类的实例，它被分配到一个变量 `p` 上。`Point` 的构造函数接受两个参数 `-1` 和 `65`，然后分别分配它们到实例变量 `x` 和 `y`（对比在 类实例（第2.3节） 中它的定义）。我们将在 new（第5.12节） 中重新审视 new 表达式的精确意思。现在，只要把它当作调用类的构造函数并返回适当的对象。

2.3.2.继承

类可以继承自其它的类，在 Haxe 中通过 extends 关键字指示：

```
class Point3 extends Point {
    var z : Int;
    public function new(x,y,z) {
        super(x,y);
        this.z = z;
    }
}
```

这个关系通常被称为 “is-a”（subsumption，包含架构，指的是类的父子继承关系）：任何 `Point3` 类的实例同时也是 `Point` 类的实例。`Point` 则作为 `Point3` 的父类，而 `Point3` 则是 `Point` 的子类。一个类可以被许多子类继承，但是只能继承一个父类。术语 “某个类的父类” 通常指它的直接父类、父类的父类，以此类推。

上面的代码和原来的 `Point` 类很相似，使用了两个新的部分：

`extends Point` 表示这个类继承自 `Point` 类。 `super(x, y)` 是调用父类的构造函数，本例中即 `Point.new` 。

在子类中定义它们自己的构造函数并不是必须的，但是如果定义了，则调用 `super()` 是必须的。不像其它一些面向对象语言，这个调用可以出现在构造函数代码中的任何地方，而不必作为构造函数中的第一个表达式。

一个类可以重写它父类的 方法（第4.3节），需要显式使用的 `override` 关键字。效果和限制将在 重写方法（第4.3.1节） 中详细介绍。

2.3.3.接口

一个接口可以被理解为一个类的签名，因为它描述了一个类的公共字段。接口不提供实现，而是单纯地提供结构化信息：

```
interface Printable {
    public function toString():String;
}
```

语法和类的相似，但有以下例外：

- 使用 `interface` 关键字而不是 `class` 关键字
- 函数不含有任何表达式
- 每个字段必须有一个显式的类型声明

接口不像 结构化子类型（第3.5.2节），它只描述与类之间的一个静态关系。一个给定的类只有当如下描述时才被认为是与给定接口兼容的：

```
class Point implements Printable { }
```

这里， `implements` 关键字表示 `Point` “是一个” `Printable` ，即每个 `Point` 的实例同时也是 `Printable` 接口的实例。虽然一个类只能有一个父类，但是它可以通过使用多个 `implements` 关键字实现多个接口：

```
class Point implements Printable
    implements Serializable
```

编译器会确保 `implements` 的假设成立。也就是说，它会确保类实现了接口所需的所有字段。当一个类或该类的父类中提供了该字段的实现时，该字段才被认为是已实现的。

接口的字段不仅限于方法，也可以是变量或者属性：

```
interface Placeable {
    public var x:Float;
    public var y:Float;
}

class Main implements Placeable {
    public var x:Float;
    public var y:Float;
    static public function main() { }
}
```

接口可以通过使用 `extends` 关键字继承多个别的接口：

```
interface Debuggable extends Printable extends Serializable
```

> Haxe 4.0.0 开始：
>
> 像类一样，接口也可以通过 `final` 关键字进行修饰以避免接口被其他接口扩展。
>
> 花絮：实现的语法 Haxe 3.0 之前 需要多个 `implements` 关键字并使用逗号 `,` 进行分隔。我们决定遵循 Java 事实上的标准，免除了逗号。这也是 Haxe 2 和 Haxe 3 之间的一个断层式的变更。

2.4.枚举实例

Haxe 提供了强大的枚举类型（简写：enum），它实际上是一个代数数据类型（ADT）。虽然它们不能有任何表达式，但用来描述数据结构是非常实用的：

```
enum Color {
    Red;
    Green;
    Blue;
    Rgb(r:Int, g:Int, b:Int);
}
```

从语义上看，这个 enum 描述了一个颜色，可能是红色，绿色，蓝色或者一个特定的 RGB 值。它的语法构成如下：

- 关键字 `enum` 表示我们要声明一个枚举
- `Color` 是枚举类型的名称，可以是任何符合类型标识符规则的字符
- 闭合的花括号中间是枚举的构造函数
- `Red`， `Green` 和 `Blue` 不需要参数
- 而 `Rgb` 需要三个 Int 类型的参数，分别是 `r`， `g` 和 `b`

Haxe 类型系统提供了一个可以和所有的枚举类型统一的类型：

> 定义 `Enum<T>`：
>
> 这个类型可以与所有的枚举类型兼容。在编译时，`Enum<T>` 可以被看作枚举类型的通用基本类型。然而，这个关系不会反应在生成的代码中。

2.4.1.Enum构造函数

类似于类和它们的构造函数，枚举类型也可以通过构造函数实例化。然而，与类不同，枚举提供了多个构造函数，可以简单地通过它们的名字进行调用：

```
var a = Red;
var b = Green;
var c = Rgb(255, 255, 0)
```

在这段代码中变量 `a`， `b` 和 `c` 的类型为 `Color`。变量 `c` 使用 `Rgb` 构造函数和参数来初始化。

所有枚举实例都可以被赋值到一个名为 `EnumValue` 的特殊类型上：

> 定义 EnumValue：
>
> `EnumValue` 是一个可以与任意 enum 实例统一的特殊类型。它被 用于 Haxe 标准库，来提供对任意枚举实例的某些操作。它也可以用于用户代码中，比如当某个 API 需要一个非特定类型的枚举实例时可以使用它。

需要着重区分枚举类型和枚举构造函数，如这个示例所展示的：

```
enum Color
{
    Red;
    Green;
```

```
    Blue;
    Rgb(r:Int,g:Int,b:Int);
}

class Main{
    public static function main(){
        var ec:EnumValue = Red; //有效
        var en:Enum<Color> = Color; //有效

        // error, Color 应该是 Enum<Color>
        // var x :Enum<Color> = Red;
    }
}
```

如果上面注释掉的行没有被注释，那么程序不会通过编译，因为 `Red` （一个枚举构造函数）不能被赋值到一个 `Enum<Color>` （一个枚举类型）类型的变量。这个关系就像类和类的实例。

> 花絮 具象化 `Enum<T>` 类型参数：
>
> 手册的一个读者困惑于上面示例中 `Color` 和 `Enum<Color>` 的区别。实际上在这里使用一个具体的类型参数是没意义的，只是为了演示目的。通常我们会在这里省略具体的类型声明，并让 类型推断（第3.6节） 来处理它。
>
> 然而，推断出的类型会不同于 `Enum<Color>` 。编译器会推断为一个含有枚举类型中的构造函数作为其 “字段” 的伪类型。自 Haxe3.2.0 起，不能在语法中表达这个类型，也没有必要这样做。

2.4.2.使用枚举

当需要用到一组值的有限集时，枚举类型是一个不错的选择。单个的构造函数表示允许的变型，并使编译器检查是否被所有可能的值所遵循：

```
enum Color{
    Red;
    Green;
    Blue;
    Rgb(r:Int, g:Int, b:Int);
}
class Main{
    public static function main(){
        var color = getColor();
        switch(color){
            case Red:
                trace("color was red");
            case Green:
                trace("color was green");
            case Blue:
                trace("color was blue");
            case Rgb(r,g,b):
                trace("color had a red value of " + r);
        }
    }

    static function getColor():Color{
        return Rgb(255, 0, 255);
    }
}
```

在取得 `getColor()` 的返回值赋值到 `color` 上的值之后， switch 语句（第5.17节） 根据 `color` 的值进行分支。前三个分支 `Red`， `Green` 和 `Blue` 此刻无关紧要，他们分别对应 `Color` 中不带参数的那几个构造函数。最后的 `Rgb(r,g,b)` 展示了如何提取构造函数中的参数：它们可以作为局部变量在 `case` 表达式体中使用，就像使用 var 表达式（第5.10节） 一样。关于进阶使用 `switch` 语句的信息将在 模式匹配（第6.4节） 章节进一步展开。

2.5.匿名结构

匿名结构可以用来组织数据，而不用显式地创建一个类型。下面的示例创建了一个结构，包含两个字段 `x` 和 `name`，并分别初始化它们的值为 `12` 和“ `foo` ”：

```
class Main {
    static public function main() {
        var myStructure = { x: 12, name: "foo"};
    }
}
```

一般语法规则如下：

- 一个包括在花括号 `{}` 内的结构
- 一个使用逗号 `,` 分隔的键值对列表
- 冒号 `:` 用来分隔的键和值，键名必须是一个有效的 标识符（第5章）
- 值可以是任何 Haxe 表达式

规则 4 意味着结构可以被嵌套和混合，如：

```
var user = {
    name : "Nicolas",
    age : 32,
    pos : [//这是一个数组
        { x : 0, y : 0 }, //这是一个匿名结构
        { x : 1, y : -1 }//这也是一个匿名结构
    ],
};
```

匿名结构的字段像类一样可以通过点号操作符 `.` 访问，如下：

```
// 获取键 name 的值，即 "Nicolas"
user.name;
// 设置 键 age 的值为 33
user.age = 33;
```

值得注意的是，使用匿名结构不会破坏类型系统。编译器确保只能访问可用的字段，也就是说，如下的程序不会编译：

```
class Test {
    static public function main() {
        var point = { x: 0.0, y: 12.0 };
        // { y : Float, x : Float } 没有字段 z
        point.z;
    }
}
```

错误信息表明编译器了解 `point` 的类型：它是一个包含了两个 `Float` 类型 `x` 和 `y` 字段的匿名结构。由于它没有字段 `z` ，访问失败。 `point` 的类型通过 类型推断（第3.6节） 被得知，这使得我们不需要为局部变量使用显式的类型声明。但是，如果 `point` 是一个字段，那么它必须通过如下形式进行声明：

```
class Path {
    var start : { x : Int, y : Int };
```

```
    var target : { x : Int, y : Int };
    var current : { x : Int, y : Int };
}
```

为了避免这种冗余的类型声明形式，尤其是对于更复杂的结构，建议使用一个 typedef（第3.1节） 关键字：

```
typedef Point = { x : Int, y : Int }

class Path {
    var start : Point;
    var target : Point;
    var current : Point;
}
```

2.5.1.结构值的JSON形式

还可以为匿名结构使用 **JavaScript** 对象记法，为键使用使用字符串字面值：

```
var point = { "x" : 1, "y" : -5 };
```

虽然允许任意的字符串字面值，但如必须是有效的 Haxe 标识符（第5章），字段才会被认为是类型的一部分。否则，Haxe 的语法不会允许对该字段访问的表达式，并且必须通过使用 `Reflect.field` 和 `Reflect.setField` 来使用反射（10.7）。

It is also possible to use **JavaScript Object Notation** for structures by using **string literals** for the keys:

```
var point = { "x" : 1, "y" : -5 };
```

While any string literal is allowed, the field is only considered part of the type if it is a valid Haxe identifier. Otherwise, Haxe syntax does not allow expressing access to such a field, and reflection has to be employed through the use of `Reflect.field` and `Reflect.setField` instead.

2.5.2. 结构类型的类记法

如果想要定义一个 结构类型 时，Haxe 允许使用在 类字段（第4章） 中所使用的语法来定义结构类型。下面的 typedef（第3.1节） 声明了一个 `Point` 类型，它包含了两个 `Int` 类型的变量字段 `x` 和 `y`：

When defining a structure type, Haxe allows the use of the same syntax described in Class Fields. The following typedef declares a `Point` type with variable fields `x` and `y` of type `Int`:

```
typedef Point = {
    var x : Int;
    var y : Int;
}
```

since Haxe 4.0.0

The fields of a structure may also be declared with `final`, which only allows them to be assigned once. Such a structure will only unify with other types if the corresponding fields are also `final`.

结构的字段也可以被修饰为 `final` ，这意味着这些字段只能被赋值一次。同时这类结构只能被统一为对应字段同样修饰为 `final` 的类型。

2.5.3 Optional Fields

结构类型的字段可以被修饰为可选的。在标准形式的声明下，通过位于字段名称前放置一个问号 `?` 进行修饰。

Fields of a structure type can be made optional. In the standard notation, this is achieved by prefixing the field name with a question mark `?`:

```
typedef User = {
  age : Int,
  name : String,
  ?phoneNumber : String
}
```

在类记法（Class Notation) 形式的声明下，则可以使用 `@:optional` 元数据来修饰可选字段

In class notation, the `@:optional` metadata can be used instead:

```
typedef User = {
  var age : Int;
  var name : String;
  @:optional var phoneNumber : String;
}
```

从 Haxe 4.0 开始，允许在类记法的结构声明中使用问号 `?` 修饰可选字段：

```
typedef User = {
  var age : Int;
  var name : String;
  var ?phoneNumber : String;
}
```

2.5.4.性能影响

通过扩展、结构子类型化（第3.5.2） 使用匿名结构，在编译为 动态类型目标语言（第2.2节） 时不会对性能产生影响。然而，对于 静态类型目标语言（第2.2节） 来说，必须执行一个动态的查找，这通常会慢于对静态字段的访问。

2.5.5.扩展

扩展被用于表示该结构具有某个给定类型的所有字段外，还具有一些额外的字段：

Extensions are used to express that a structure has all the fields of a given type as well as some additional fields of its own:

```
typedef IterableWithLength<T> = {
  > Iterable<T>,
  // read only property
  var length(default, null):Int;
```

```
}

class Main {
  static public function main() {
    var array = [1, 2, 3];
    var t:IterableWithLength<Int> = array;
  }
}
```

大于操作符 `>` 被用于表示创建一个 `Iterable<T>` 的扩展。然后补充额外的类字段，此例中，需要一个 `Int` 类型的 只读 属性 `length` 。

The greater-than operator `>` denotes that an extension of `Iterable<T>` is being created, with the additional class fields following. In this case, a read-only property `length` of type `Int` is required.

为了能与 `IterableWithLength<T>` 兼容，一个类必须能够与 `Iterable<T>` 兼容的同时提供一个 只读 的 `Int` 类型属性 `length` 。以上例子中赋值了一个 `Array` ，其刚好满足这两点要求。

In order to be compatible with `IterableWithLength<T>` , a type must be compatible with `Iterable<T>` and provide a read-only `length` property of type `Int` . The previous example assigns an `Array` , which happens to fulfill these requirements.

since Haxe 3.1.0

多个结构可以被同时扩展：

Multiple structures can be extended at once:

```
typedef WithLength = {
  var length(default, null):Int;
}

typedef IterableWithLengthAndPush<T> = {
  > Iterable<T>,
  > WithLength,
  function push(a:T):Int;
}

class Main {
  static public function main() {
    var array = [1, 2, 3];
    var t:IterableWithLengthAndPush<Int> = array;
  }
}
```

since Haxe 4.0.0

有一个额外的记法可以被用于扩展，通过以 `&` 符号分格每一个欲扩展的结构来表示。

An alternative notation for extension can be used, denoted by separating each extended structure with an `&` symbol.

```
typedef Point2D = {
  var x:Int;
  var y:Int;
}

typedef Point3D = Point2D & {z:Int};

class Main {
```

```
  static public function main() {
    var point:Point3D = {x: 5, y: 3, z: 1};
  }
}
```

补充另一例子：

```
typedef IterableWithLengthAndPush<T> = {
  > Iterable<T>,
  > WithLength,
  function push(a:T):Int;
}
//可以改写为：
typedef IterableWithLengthAndPush<T> =
Iterable<T> & WithLength &
{
    function push(a:T):Int;
}
```

2.6.函数类型

函数类型与 单形（第2.9节） 都是一个类型，尽管他们无处不在，但是这两种类型都被很好的隐藏了起来。我们可以使用 一个特殊的 Haxe 标识符 `$type` 来使得隐藏其后类型浮出水面，它可以在编译时输出它所持表达式的类型：

```
class Main {
    static public function main() {
        $type(test); // i : Int -> s : String -> Bool
        $type(test(1, "foo")); // Bool
    }
    static function test(i:Int, s:String):Bool {
        return true;
    }
}
```

第一条 `$type` 表达式的输出与 `test` 的函数声明十分相似，同时也有微妙的区别：函数参数被特殊的箭头符号 `->` 而不是逗号 `,` 所分隔，且函数返回类型出现在结尾的另一个 `->` 符号之后。

不论哪个记法，很明显，函数 `test` 接受第一个 Int 类型的参数，第二个 String 类型的参数，并返回一个 Bool 类型值。如果调用这个函数，例如 `test(1,"foo")`，并将其放在第二个 `$type` 语句中，Haxe 类型检查器将会检查 `1` 是否可以被赋值到 `Int` 类型参数，然后检查 `"foo"` 是否可以被赋值到 `String` 类型参数，检查通过之后函数调用的结果就会和 `test` 函数返回值的类型相同，即，一个 `Bool` 类型。

如果一个函数类型有其它函数类型的参数或者返回值，则可以使用括号对它们进行正确的分组。例如，`Int->(Int->Void)->Void` 表示一个函数，第一个参数为 Int 类型，第二个参数是函数 `Int->Void` 类型，返回 `Void`。

2.6.1.可选参数

可选参数通过在参数标识符前面前置一个问号 `?` 来声明：

```
class Main {
    static public function main() {
        // ?i : Int -> ?s : String -> String
        $type(test);
        trace(test()); // i: null, s: null
        trace(test(1)); // i: 1, s: null
        trace(test(1, "foo")); // i: 1, s: foo
        trace(test("foo")); // i: null, s: foo
    }
    static function test(?i:Int, ?s:String) {
        return "i: " +i + ", s: " +s;
    }
}
```

函数 `test` 有两个可选参数：Int 类型的 `i` 和 String 类型的 `s`。这在第3行的函数类型输出直接反映出来。这个例子程序调用了4次 `test`，并打印出它的返回值：

- 第一次调用不带任何参数
- 第二次调用带有一个单独的参数 `1`
- 第三次调用带了两个参数 `1` 和 "`foo`"
- 第四次调用只有单独的参数“`foo`”

输出内容显示，函数调用中被省略的可选参数的值为 `null` 。这意味着这些参数的类型必须能够接受 `null` 作为其值，这个问题在 为空性（第2.2节） 中详细讨论。Haxe 编译器会确保当编译至 静态目标平台（第2.2节）时，基本类型的可选参数被推断为 `Null<T>` 使得其成为一个 “可空的” 类型。

前三个调用非常直观，第四个调用可能显得有些意外：如果当前位置提供的值可以被赋值给下一个位置的参数时，则允许跳过可选参数直接赋值给下一个位置的参数。

补充：跳过可选参数的规则是可传递的，比如

```
static function test(?i:Int, ?f:Float, ?s:String)
{
    trace(s);
}
//当调用 test("foo"); 时将会输出 “foo”
```

2.6.2.默认值

Haxe 允许通过分配一个常量值来为参数提供默认值，：

```
class Main {
    static public function main() {
        // ?i : Int -> ?s : String -> String
        $type(test);
        trace(test()); // i: 12, s: bar
        trace(test(1)); // i: 1, s: bar
        trace(test(1, "foo")); // i: 1, s: foo
        trace(test("foo")); // i: 12, s: foo
    }
    static function test(?i = 12, s = "bar") {
        return "i: " +i + ", s: " +s;
    }
}
```

这个示例和前面 可选参数（第2.6.1节） 中的非常相似，唯一的区别是函数的参数 `i` 和 `s` 分别被赋值为 `12` 和 “ `bar` ”。效果为以默认值取代 `null` ，可以在调用时省略某个参数。Haxe 中的默认值并不是类型的一部分，而且不能在调用时更改函数的默认值（意指只应用于当次调用，而不会改变函数的默认值定义。除非函数是 内联（第4.4.2节） 的，被认为是一个比较典型的处理）。在一些目标语言中，编译器仍可能会传递 `null` 值作为省略的参数值，生成类似于下面所示的代码到函数中：

```
static function test(i = 12, s = "bar") {
    if (i == null) i = 12;
    if (s == null) s = "bar";
    return "i: " +i + ", s: " +s;
}
```

在对性能有要求的代码中这个问题需要被考虑到，因为没有默认值的解决方案有时可能更可行。

2.7.动态类型

虽然 Haxe 有一个静态的类型系统，但这个类型系统实际上可以通过使用 Dynamic 类型关闭。一个 动态值可以被赋值给任何类型；而任何值都可以被分配到动态类型。这有一些缺点：

- 编译器将不会再在接受指定类型的赋值、函数调用和其它构造中进行类型检查。
- 某些优化，特别是编译为静态目标语言时，将不能再被使用。
- 一些常见的错误，例如字段访问中的一个拼写错误，将不能在编译时被发现，可能会引发运行时的错误。
- 如果字段是通过 Dynamic 类型使用，无用代码消除（第8.2节） 不能检测使用到的字段

使用 Dynamic 类型可能引发运行时错误的例子非常容易出现。思考下面的两行代码到静态目标语言的编译：

```
var d:Dynamic = 1;
d.foo;
```

尝试在 Flash 播放器运行编译后的程序，会产生一个错误 ，属性 `foo` 在 数值类型中没有找到，而且没有默认值。不使用 Dynamic，这会在编译时被侦测到。

> 花絮：Haxe 3之前的 Dynamic 类型推断 Haxe 3 编译器从不推断一个类型为 Dynamic 类型，所以用户必须明确它。之前的 Haxe 版本曾经推断 数组为一个混合类型，如 `[1, true, "foo"]` 为 `Array`。我们发现这个行为会引发太多的类型问题，因此在 Haxe 3 中移除了它。

应该尽量少的使用 Dynamic 类型，因为很多情况下都有更好的选择，但是有时候实际会用到它。Haxe 反射（第10.7节） API中部分使用了 Dynamic 类型，而且有时候它是在处理编译时未知的自定义数据结构最好选择。当被用一个 单形（第2.9节）统一（第3.5节） 的时候，Dynamic 类型以一种特殊的方式运行。单形没有绑定到 Dynamic，这可以在如下例子中带来令人惊喜的结果：

```
class Main {
    static function main() {
        var jsonData = '[1, 2, 3]';
        var json = haxe.Json.parse(jsonData);
        $type(json); // Unknown<0>
        for (i in 0...json.length) {
            // Array access is not allowed on
            // {+ length : Int }
            trace(json[0]);
        }
    }
}
```

尽管 `Json.parse` 的返回类型是 Dynamic 类型，局部变量 `json` 的类型并没有绑定到动态类型，仍然保持了一个单形。然后它在 json.length 字段访问上被推断为一个 匿名结构（第2.5节），使后面的 `json[0]` 数组访问失败。为了避免这个问题，变量 `json` 可以通过使用 `var json:Dynamic` 显式的声明为 Dynamic 类型。

> 花絮：标准库中的 Dynamic 类型 Dynamic 类型在 Haxe 3 之前非常频繁的使用在标准库中。随着 Haxe 类型系统的持续改进，Dynamic 类型的出现在通往 Haxe 3的版本被减少。

2.7.1.Dynamic使用类型参数

Dynamic 是一个特殊的类型，因为它允许使用和不使用一个 类型参数（第3.2节） 来进行显式的声明。如果这样一个类型参数被提供，Dynamic（第2.7节） 中描述的 语义被限制为所有的字段兼容该参数类型：

```
var att : Dynamic<String> = xml.attributes;
// 有效，值为一个 String 类型
att.name = "Nicolas";
// dito (这个文档太旧了)
att.age = "26";
// error,值不是 String 类型
att.income = 0;
```

2.7.2.实现Dynamic

类可以 实现（第2.3.3节） Dynamic 类型，和提供任意字段访问的 Dynamic 。前一种情况，字段可以有任何类型，而后一种，它们被限制兼容参数类型：

```
 class ImplementsDynamic
    implements Dynamic<String> {
        public var present:Int;
        public function new() {}
}
class Main {
    static public function main() {
        var c = new ImplementsDynamic();
        // 有效的，present 是一个存在的字段
        c.present = 1;
        // 有效，分配的值是一个 String
        c.stringField = "foo";
        // 错误，Int应该是 String
        //c.intField = 1;
    }
}
```

实现 Dynamic 不符合实现其它接口的需求。预期的字段仍然必须被明确实现。实现 Dynamic 的类（带或者不带类型参数）也可以利用一个特别的方法名字叫做 `resolve` 。如果一个 读访问（第4.2节） 被做出，而且被讨论的字段不存在， `resolve` 方法被调用，并以这个字段的名字作为参数：

```
class Resolve implements Dynamic<String> {
    public var present:Int;
    public function new() {}
    function resolve(field:String) {
        return "Tried to resolve " +field;
    }
}

class Main {
    static public function main() {
        var c = new Resolve();
        c.present = 2;
        trace(c.present);
        trace(c.resolveMe);
    }
}
```

2.8.抽象类型

一个抽象类型在运行时实际上是一个不同的类型。它是一个编译时功能，在固有类型之上定义来修改或者增强它们的行为的类型：

```
abstract AbstractInt(Int) {
    inline public function new(i:Int) {
        this = i;
    }
}
```

从这个例子我们可以得出以下几点：

- 关键字 `abstract` 表示我们声明一个抽象类型
- `AbstractInt` 是抽象类型的名称，可以是任何符合类型标识符规则的字符
- 圆括号 `()` 中的是潜在的类型 Int
- 大括号 `{}` 中的是字段
- 构造函数 `new` 接受一个 Int 类型的参数 `i`

潜在类型

> 定义：潜在类型 抽象类型的潜在类型是用来代表抽象类型在运行时的类型。通常是一个具体的（即非抽象的）类型，但是也可以是另一个抽象类型。

这个语法让人联想到类，语义上它们事实的确非常相似。实际上，每个在抽象类型“体”中的（即所有花括号之后的一切）都被解析为类字段。抽象类型可以有 方法（第4.3节） 字段和 非物理（第4.2.3节）属性（第4.2节） 字段。此外，抽象类型可以被像类一样实例化和使用：

```
class Main {
    static public function main() {
        var a = new AbstractInt(12);
        trace(a); //12
    }
}
```

如前所述，抽象类型是一个编译时功能，所以看看上面示例实际生成的内容会很有趣。一个合适的目标是 JavaScript，它往往可以生成简洁干净的代码。编译上面的代码（使用 `haxe -main MyAbstract -js myabstract.js` ）会显示如下 JavaScript 代码：

```
var a = 12;
console.log(a);
```

抽象类型 `Abstract` 在输出中完全消失了，剩下的只是一个它潜在类型的值，Int。因为 `Abstract` 的构造函数是内联的 （在 内联部分（第4.4.2节） 我们将进行学习的内容 ） ，它的内联表达式分配一个值到这里。当以类进行思考的话，这可能是令人惊讶的。然而，这恰巧是我们希望在抽象类型的上下文中表达的。抽象类型的任何内联成员方法都可以分配到这里，从而修改 “内部的值”。在这点上，一个好的问题是 “如果一个成员函数没有被内联声明将发生什么”，因为代码显然必须放到某个地方。Haxe 创建一个私有类，即已知的实现类，它将所有的抽象成员函数作为接受一个附加的类型为潜在类型的首参数的静态函数。虽然技术上上这是一个实现细节，但它可以被用于 选择函数（第2.8.4节）。

> 花絮：基本类型和抽象类型 在抽象类型到来之前，所有基本类型都实现为外部类或者枚举。虽然这很好的考虑了某些方面，如 Int 是 Float 的一个“子类”，但这也在别处引起问题。例如，通过 Float 作为一个外部类，它会和空的结构 `{}` 统一，使得不可能限制一个类型只接受真正的对象。

An abstract type is a type which is actually a different type at run-time. It is a compile-time feature which defines types "over" concrete types in order to modify or augment their behavior:

```
abstract AbstractInt(Int) {
  inline public function new(i:Int) {
    this = i;
  }
}
```

We can derive the following from this example:

- The keyword `abstract` denotes that we are declaring an abstract type.
- `AbstractInt` is the name of the abstract type and could be anything conforming to the rules for type identifiers.
- The **underlying type** `Int` is enclosed in parentheses `()`.
- The fields are enclosed in curly braces `{}`,
- which are a constructor function `new` accepting one argument `i` of type `Int`.

> **Define: Underlying Type**
>
> The underlying type of an abstract is the type which is used to represent said abstract at runtime. It is usually a concrete (i.e. non-abstract) type but could be another abstract type as well.

The syntax is reminiscent of classes and the semantics are indeed similar. In fact, everything in the "body" of an abstract (everything after the opening curly brace) is parsed as class fields. Abstracts may have method fields and non-physical property fields.

Furthermore, abstracts can be instantiated and used just like classes:

```
class Main {
  static public function main() {
    var a = new AbstractInt(12);
    trace(a); // 12
  }
}
```

As mentioned before, abstracts are a compile-time feature, so it is interesting to see what the above actually generates. A suitable target for this is JavaScript, which tends to generate concise and clean code. Compiling the above using `haxe --main MyAbstract --js myabstract.js` shows this JavaScript code:

```
var a = 12;
console.log(a);
```

The abstract type `Abstract` completely disappeared from the output and all that is left is a value of its underlying type, `Int`. This is because the constructor of `Abstract` is inlined - something we shall learn about later in the section Inline - and its inlined expression assigns a value to `this`. This might be surprising when thinking in terms of classes. However, it is precisely what we want to express in the context of abstracts. Any **inlined member method** of an abstract can assign to `this` and thus modify the "internal value".

One problem may be apparent - what happens if a member function is not declared inline? The code obviously must be placed somewhere! Haxe handles this by creating a private class, known as the **implementation class**, which contains all the abstract member functions as static functions accepting an additional first argument `this` of the underlying type.

> **Trivia: Basic Types and abstracts**
>
> Before the advent of abstract types, all basic types were implemented as extern classes or enums. While this nicely took care of some aspects such as `Int` being a "child class" of `Float`, it caused issues elsewhere. For instance, with `Float` being an extern class, it would unify with the empty structure `{}`, making it impossible to constrain a type to accept only real objects.

2.8.1.隐式类型转换

与类不同，抽象类型允许定义隐式转换。有两种类型的隐式转换：

直接转换： 允许抽象类型和其它类型之间的直接转换。这通过从或者往抽象类型添加规则来定义，而且只允许用于和抽象类型的潜在类型统一的类型。

类字段： 允许通过调用特别的转换函数进行转换。这些函数通过使用 `@:to` 和 `@:from` 元数据定义。这种类型的转换允许用于所有类型。

如下的代码示例展示直接转换的一个例子：

```
abstract MyAbstract(Int) from Int to Int {
    inline function new(i:Int) {
        this = i;
    }
}

class Main {
    static public function main() {
        var a:MyAbstract = 12;
        var b:Int = a;
    }
}
```

我们声明 `MyAbstract` 可以是从或者到 Int 类型，意思是它可以被用 Int 分配，并且可以分配到 Int 类型。这在第9和10行展示，第一个分配 Int `12` 到 `MyAbstract` 类型的变量 `a`（通过使用 `from Int` 声明），然后这个抽象类型分配回 Int 类型变量 `b`（通过使用 `to Int` 声明）。

另一种的类字段转换有相同的语义，但是定义完全不同：

```
abstract MyAbstract(Int) {
    inline function new(i:Int) {
        this = i;
    }
    @:from
    static public function fromString(s:String) {
        return new MyAbstract(Std.parseInt(s));
    }

    @:to
    public function toArray() {
        return [this];
    }
}
```

```
class Main {
    static public function main() {
        var a:MyAbstract = "3";
        var b:Array<Int> = a;
        trace(b); // [3]
    }
}
```

通过添加 `@:from` 到一个静态函数，这个函数获得资格作为从它的参数类型到抽象类型的隐式转换函数。这些函数必须返回一个抽象类型的值。它们还必须被声明为静态的：

类似的，添加 `@:to` 到一个函数，它获取资格作为从抽象类型到它的返回类型的隐式转换函数。这些函数通常是成员函数，但是它们可以被声明为静态，然后作为一个 选择函数（第2.8.4节）。

在例子中，方法 `fromString` 允许分配值“`3`”到 `MyAbstract` 类型的变量 `a`，而方法 `toArray` 允许分配抽象类型到 `Array` 类型的变量 `b`。

当使用这种类型的转换，转换函数的调用被插入在需要的地方。当查看 JavaScript 输出的时候会很明显：

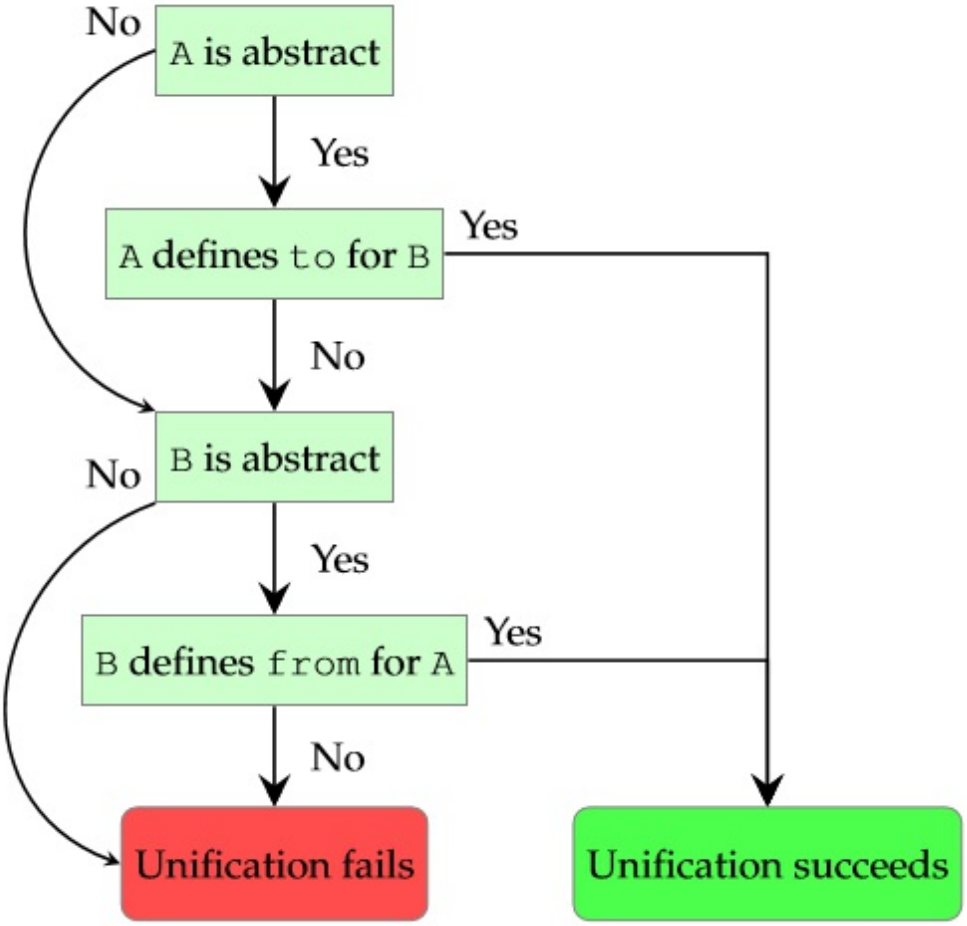
```
var a = _ImplicitCastField.MyAbstract_Impl_.fromString("3");
var b = _ImplicitCastField.MyAbstract_Impl_.toArray(a);
```

这可以被进一步优化，通过 内联（第4.4.2节） 两种转换函数，使得输出如下：

```
var a = Std.parseInt("3");
var b = [a];
```

当分配一个类型 A 到一个类型 B，并且至少它们中的一个为抽象类型时，选择的规则很简单：

- 1.如果 A 不是一个抽象类型，去到 3
- 2.如果 A 定义了一个允许到 B 的 `to` 转换，去到6
- 3.如果 B 不是一个抽象类型，去到 5
- 4.如果 B 定义一个允许到 A 的 `from` 转换，去到6
- 5.停止，统一失败
- 6.停止，统一成功



经过设计，隐式转换不被传递，就像下面的例子展示的：

```
abstract A(Int) {
    public function new() this = 0;
    @:to public function toB() return new B();
}

abstract B(Int) {
    public function new() this = 0;
    @:to public function toC() return new C();
}

abstract C(Int) {
    public function new() this = 0;
}

class Main {
    static public function main() {
        var a = new A();
        var b:B = a; // valid, uses A.toB
        var c:C = b; // valid, uses B.toC
        var c:C = a; // error, A should be C
    }
}
```

虽然从 A 到 B 的转换和从 B 到 C 的转换分别是允许的，而一个传递的从 A 到 C 的转换则不允许。这是为了避免不明确的转换路径，并保持一个简单的选择规则。

Unlike classes, abstracts allow defining implicit casts. There are two kinds of implicit casts:

- Direct: Allows direct casting of the abstract type to or from another type. This is defined by adding `to` and `from` rules to the abstract type and is only allowed for types which unify with the underlying type of the abstract.
- Class field: Allows casting via calls to special cast functions. These functions are defined using `@:to` and `@:from` metadata. This kind of cast is allowed for all types.

The following code example shows an example of **direct** casting:

```
abstract MyAbstract(Int) from Int to Int {
  inline function new(i:Int) {
    this = i;
  }
}

class Main {
  static public function main() {
    var a:MyAbstract = 12;
    var b:Int = a;
  }
}
```

We declare `MyAbstract` as being `from Int` and `to Int`, appropriately meaning it can be assigned from `Int` and assigned to `Int`. This is shown in lines 9 and 10, where we first assign the `Int` `12` to variable `a` of type `MyAbstract` (this works due to the `from Int` declaration) and then that abstract back to variable `b` of type `Int` (this works due to the `to Int` declaration).

Class field casts have the same semantics, but are defined completely differently:

```
abstract MyAbstract(Int) {
  inline function new(i:Int) {
    this = i;
  }

  @:from
  static public function fromString(s:String) {
    return new MyAbstract(Std.parseInt(s));
  }

  @:to
  public function toArray() {
    return [this];
  }
}

class Main {
  static public function main() {
    var a:MyAbstract = "3";
    var b:Array<Int> = a;
    trace(b); // [3]
  }
}
```

By adding `@:from` to a static function, that function qualifies as an implicit cast function from its argument type to the abstract. These functions must return a value of the abstract type. They must also be declared `static`.

Similarly, adding `@:to` to a function qualifies it as implicit cast function from the abstract to its return type.

In the previous example, the method `fromString` allows the assignment of value `"3"` to variable `a` of type `MyAbstract` while the method `toArray` allows assigning that abstract to variable `b` of type `Array<Int>`.

When using this kind of cast, calls to the cast functions are inserted where required. This becomes obvious when looking at the JavaScript output:

```
var a = _ImplicitCastField.MyAbstract_Impl_.fromString("3");
var b = _ImplicitCastField.MyAbstract_Impl_.toArray(a);
```

This can be further optimized by inlining both cast functions, turning the output into the following:

```
var a = Std.parseInt("3");
var b = [a];
```

The **selection algorithm** when assigning a type `A` to a type `B` where at least one is an abstract is simple:

1. If `A` is not an abstract, go to 3.
2. If `A` defines a **to**-conversion that admits `B`, go to 6.
3. If `B` is not an abstract, go to 5.
4. If `B` defines a **from**-conversion that admits `A`, go to 6.
5. Stop, unification fails.
6. Stop, unification succeeds.

By design, implicit casts are **not transitive**, as the following example shows:

```
abstract A(Int) {
  public function new()
    this = 0;

  @:to public function toB() return new B();
}

abstract B(Int) {
  public function new()
    this = 0;

  @:to public function toC() return new C();
}

abstract C(Int) {
  public function new()
    this = 0;
}

class Main {
  static public function main() {
    var a = new A();
    var b:B = a; // valid, uses A.toB
    var c:C = b; // valid, uses B.toC
    var c:C = a; // error, A should be C
  }
}
```

While the individual casts from `A` to `B` and from `B` to `C` are allowed, a transitive cast from `A` to `C` is not. This is to avoid ambiguous cast paths and retain a simple selection algorithm.

2.8.2.运算符重载

抽象类型通过添加 `@:op` 元数据到类字段，允许一元和二元运算符的重载：

```
abstract MyAbstract(String) {
    public inline function new(s:String) {
        this = s;
    }

    @:op(A * B)
    public function repeat(rhs:Int):MyAbstract {
        var s:StringBuf = new StringBuf();
        for (i in 0...rhs)
        s.add(this);
        return new MyAbstract(s.toString());
    }
}

class Main {
    static public function main() {
        var a = new MyAbstract("foo");
        trace(a * 3); // foofoofoo
    }
}
```

通过默认的 `@:op(A*B)`，当左面的值类型是 `MyAbstract` 而且右侧值是 Int 类型的时候，函数 `repeat` 作为乘法 `*` 运算符的运算符。用法在第17行显示，编译到 JavaScript 之后代码成为这样：

```
console.log(_AbstractOperatorOverload.MyAbstract_Impl_.repeat(a,3));
```

类似于通过 类字段（第2.8.1） 方式隐式转换，重载之后的方法的调用被插入到需要的地方。

示例中的 `repeat` 函数是不可交换的：当 `MyAbstract * Int` 工作， `Int * MyAbstract` 则不工作。如果这应该同时允许，可以添加 `@:commutative` 元数据。如果它只可以为 `Int * MyAbstract` 工作，而不是为 `MyAbstract * Int`，重载方法可以被设置为 `static`，接受 Int 和 MyAbstract 分别作为第一个和第二个类型。

重载一元运算符是相似的：

```
abstract MyAbstract(String) {
    public inline function new(s:String) {
        this = s;
    }

    @:op(++A) public function pre()
        return "pre" + this;
    @:op(A++) public function post()
        return this + "post";
}

class Main {
    static public function main() {
        var a = new MyAbstract("foo");
        trace(++a); // prefoo
        trace(a++); // foopost
    }
}
```

二元运算和一元运算符重载都可以返回任何类型。

暴露潜在类型的操作 还可以省略 `@:op` 函数的方法体，但是只有抽象类型的潜在类型允许涉及的操作，并且结果类型可以被赋值回抽象类型的时候。

```
abstract MyAbstractInt(Int) from Int to Int {
    // 下面一行从潜在类型Int暴露 （A>B）操作符
    // 注意，并没有使用函数体
    @:op(A > B) static function gt( a:MyAbstractInt, b:MyAbstractInt ) : Bool;
}

class Main {
    static function main() {
        var a:MyAbstractInt = 42;
        if(a > 0) trace('Works fine, > operation implemented!');

        // 小于操作符没有实现
        // 这会引起一个 “不能对比MyAbstractInt 和 Int”的错误:
        if(a < 100) { }
    }
}
```

2.8.3.数组访问

数组访问描述了特定的语法，传统上用于访问数组中的某个偏移量的值。通常只允许带有 Int 类型的参数。然而，使用抽象类型也可以定义自己的数组访问方法。Haxe 标准库（第10章） 通过它的 Map 类型应用这种方式，其中下面两个方法可以被发现：

```
@:arrayAccess
public inline function get(key:K) {
    return this.get(key);
}

@:arrayAccess
public inline function arrayWrite(k:K, v:V):V {
    this.set(k, v);
    return v;
}
```

有两种数组访问方法：

- 如果一个 `@:arrayAccess` 方法接受一个参数，它是一个 `getter`；
- 如果一个 `@:arrayAccess` 方法接受两个参数，它是一个 `setter`。

然后下面看到的 `get` 和 `arrayWrite` 方法允许这样使用：

```
class Main {
    public static function main() {
        var map = new Map();
        map["foo"] = 1;
        trace(map["foo"]);
    }
}
```

在这一点上，看到数组访问字段的调用被插入到输入内容应该不会太让人吃惊：

```
 map.set("foo",1);
 console.log(map.get("foo")); // 1
```

数组访问解析的顺序 由于Haxe 3.2 版本之前的一个 bug，检查 `:arrayAccess` 字段的顺序是未被定义的。在 3.2 版本中已经修复所以现在总是从上到下进行检查字段：

```
abstract AString(String) {
    public function new(s) this = s;
    @:arrayAccess function getInt1(k:Int) {
        return this.charAt(k);
    }
    @:arrayAccess function getInt2(k:Int) {
        return this.charAt(k).toUpperCase();
    }
}

class Main {
    static function main() {
        var a = new AString("foo");
        trace(a[0]); // f
    }
}
```

数组访问 `a[0]` 被解析到 `getInt1` 字段，使得小写的 `f` 被返回。结果可能和 Haxe 3.2 之前的版本不同。

先定义的字段有优先级，即使它们需要一个 隐式的转换（第2.8.1节）。

2.8.4.选择函数

由于编译器提升抽象成员函数为静态函数，可以手动定义静态函数并使用它们到一个抽象类的实例。这里的语法和那些第一个函数参数类型决定函数被定义为什么类型的 静态扩展（第6.3节） 类似，：

```
abstract MyAbstract<T>(T) from T {
    public function new(t:T) this = t;

    function get() return this;
    @:impl
    static public function getString(v:MyAbstract<String>):String {
        return v.get();
    }
}

class Main {
    static public function main() {
        var a = new MyAbstract("foo");
        a.getString();
        var b = new MyAbstract(1);
        // Int should be MyAbstract<String>
        b.getString();
    }
}
```

抽象类型 `MyAbstract` 的方法 `getString` 被定义为接受一个 `MyAbstract` 类型首参数。这使它在第14行可以用在变量 `a` 上（因为 `a` 的类型为 `MyAbstract` ），但是不能用在变量 `b` 上， `b` 的类型是 `MyAbstract` 。

> 花絮：意外的功能 选择函数是被发现的，而不是真的设计了它的用法。在第一次提到这个想法后，只需要编译器中的一点调整就可以使它们工作。它们的这些发现还引入了多类型抽象类型，比如 Map 。

2.8.5.枚举抽象类型

Haxe 3.0版本之后

通过添加 `:enum` 元数据到一个抽象类型的定义，这个抽象类可以被用来定义有限值的集合：

```
@:enum
abstract HttpStatus(Int) {
    var NotFound = 404;
    var MethodNotAllowed = 405;
}

class Main {
    static public function main() {
        var status = HttpStatus.NotFound;
        var msg = printStatus(status);
    }

    static function printStatus(status:HttpStatus) {
        return switch(status) {
            case NotFound:
                "Not found";
            case MethodNotAllowed:
                "Method not allowed";
        }
    }
}
```

Haxe 编译器使用它们的值替换所有的字段访问到 `HttpStatus` 抽象类型，在 JavaScript 的输出则很明显：

```
 Main.main = function() {
    var status = 404;
    var msg = Main.printStatus(status);
};
Main.printStatus = function(status) {
    switch(status) {
        case 404:
            return "Not found";
        case 405:
            return "Method not allowed";
    }
};
```

这和访问 内联（第4.4.2节） 声明的变量很相似，但是有几个优点：

- 类型工具可以确保集合的所有值会被正确分配类型。
- 匹配模式在 匹配（第6.4节） 一个枚举抽象类型时检查其 穷尽性（第6.4.10节）。
- 用更少的语句定义字段。

2.8.6.转发抽象类型字段

Haxe 3.0版本之后

当包装一个潜在类型，有时候需要保持部分它的功能。因为手工编写转发函数非常繁琐，Haxe 允许添加 `:forward` 元数据到一个抽象类型：

```
@:forward(push, pop)
abstract MyArray<S>(Array<S>) {
    public inline function new() {
        this = [];
```

```
    }
}

class Main {
    static public function main() {
        var myArray = new MyArray();
        myArray.push(12);
        myArray.pop();
        // MyArray<Int> has no field length
        //myArray.length;
    }
}
```

这个例子中的 `MyArray` 抽象类封装了 Array。它的 `:forward` 元数据有两个参数，对应要被转发到潜在类型的字段名。在这个例子中， `main` 方法实例化 `MyArray` 并访问它的 `push` 和 `pop` 方法。注释行表明 `length` 字段是不可用的。

像往常一样我们可以查看 JavaScript 输出来看一下代码如何生成：

```
Main.main = function() {
    var myArray = [];
    myArray.push(12);
    myArray.pop();
};
```

也可以使用 `:forward` 而不带任何参数，来转发所有字段。当然 Haxe 编译器仍然保证字段实际上存在于潜在类型。

> 花絮：实现为宏 `:enum` 和 `:forward` 功能都是最初被使用 构建宏（第9.5节） 实现的。虽然这可以在非宏代码很好的运行，但是如果这些功能从宏内部运行它会引起问题。这个实现后来被移到编译器中。

2.8.7.核心类型抽象

Haxe 标准库定义了一组基础类型作为核心类型抽象。它们通过 `:coreType` 元数据识别，而且缺失一个潜在类型的声明。这些抽象类型仍然可以被理解为表示不同的类型。不过，这个类型是 Haxe 目标语言原生的。

引入自定义核心类型抽象在用户代码中是很有必要的，因为它需要 Haxe 目标语言可以理解它的意思。然而，对于宏的作者和新的 Haxe 目标语言可能是很有趣的用例。

与难懂的抽象类型的相比，核心类型抽象有下面的属性：

- 它们没有潜在类型。
- 它们被认为是可空的，除非带有 `:notNull` 元数据的注解。
- 它们被允许没有表达式形式的 数组访问（第2.8.3节） 函数声明。
- 没有表达式的 运算符重载字段（第2.8.2节） 不会被强制追随 Haxe 的类型语法形式。

2.9.单形

一个单形（Monomorph）会在 一致性检查（3.5） 时变形（morph）为另外一个类型。有关这一类型的进一步细节将在 类型推断（3.6）中进行解释。

A monomorph is a type which may, through unification(3.5), morph into a different type later. Further details about this type are explained in the section on type inference(3.6)

3.类型系统

我们在 类型（第2章） 中已经学习了不同的类型，现在来看看它们之间如何交互。我们先简单介绍一下 typedef（第3.1节），这是一种给一个复杂的类型命名（或别名）的机制。有许多用法，其中当处理有类型参数（第3.2节）的类型时，使用 typedef 会十分方便。

> *We learned about the different kinds of types in Types and it is now time to see how they interact with each other. We start off easy by introducing typedef, a mechanism to give a name (or alias) to a more complex type. Among other use cases, typedefs will come in handy when working with types that have type parameters.*

通过检查两个给定类型之间是否兼容，类型安全性有了显著提升。这意味着编译器会尝试对它们执行一致性检查，更多细节将在 一致性检查（第3.5节）中展开。

> *A significant amount of type-safety is achieved by checking if two given types are compatible. Meaning, the compiler tries to perform **unification** between them as detailed in Unification.*

所有的类型以模块的形式组织，且可以通过路径进行定位。 模块和路径（第3.7节） 中会详细解释相关的机制。

> *All types are organized in **modules** and can be addressed through **paths**. Modules and Paths will give a detailed explanation of the related mechanics.*

3.1.Typedef

在讨论 匿名结构（第2.5节） 的时候我们简要的介绍了如何通过关键字 `typedef` 给定一个名字来缩短一个复杂的结构类型（第2.5节）的声明。这正是 `typedef` 所擅长的事情。给匿名结构类型命名甚至可以认为是它们的主要用途。事实上，这种方式的使用是如此的常见以至于许多 Haxe 用户甚至以为 `typedef` 实际上就是结构类型。

> *We briefly looked at typedefs while talking about anonymous structures and saw how we could shorten a complex structure type by giving it a name. This is precisely why typedefs are useful. Giving names to structure types might even be considered their primary function, and is so common that the distinction between the two appears somewhat blurry. Many Haxe users consider typedefs to actually* ***be*** *the structure.*

`typedef` 可以给任意类型命名：

> *A typedef can give a name to any other type:*

```
typedef IA = Array<Int>;
```

这使我们能够使用 `IA` 到我们需要使用 `Array<Int>` 的地方。虽然这只不过节省了几次敲键盘的次数，但在用于更复杂、复合度更高的类型时会产生很大的差异。而这也是为什么 `typedef` 和结构配合使用时显得如此连贯的原因：

> *This enables us to use `IA` in places where we would normally use `Array<Int>`. While this saves only a few keystrokes in this particular case, it can make a larger difference for more complex, compound types. Again, this is why typedef and structures seem so connected:*

```
typedef User = {
    var age : Int;
    var name : String;
}
```

一个 `typedef` 并不是一个文本的替换，而是一个实实在在的类型。它甚至可以有 类型参数（第3.2节），比如 Haxe 标准库中的 `Iterable` 类型所展示的：

> *Typedefs are not textual replacements, but are actually real types. They can even have type parameters as the `Iterable` type from the Haxe Standard Library demonstrates:*

```
typedef Iterable<T> = {
   function iterator() : Iterator<T>;
}
```

3.2.类型参数

Haxe允许许多类型的参数化，就像类字段（第4章）和枚举构造函数（第2.4.1节）。类型参数通过闭合的尖括号囊括以逗号分隔的类型参数名来定义。一个简单的例子来自于Haxe标准库，就是 Array：

Haxe allows parametrization of a number of types, as well as class fields (4) and enum constructors (2.4.1). Type parameters are defined by enclosing comma-separated type parameter names in angle brackets <>. A simple example from the Haxe Standard Library is Array:

```
class Array<T> {
   function push(x : T) : Int;
}
```

每当一个Array的实例被创建，它的类型参数 T 成为一个 单形（第2.9节）。也就是说，它可以被绑定到任何类型，但是一次只有一个。可以是显式地触发绑定，通过调用构造函数并显式地提供类型（ `new Array<String>()` ），或者隐式地触发，通过类型推断（第3.6节），例如当我们调用 `arrayInstance.push("foo")` 。

Whenever an instance of Array is created, its type parameter T becomes a monomorph (2.9). That is, it can be bound to any type, but only one at a time. This binding can happen explicitly by invoking the constructor with explicit types (new Array()) or implicitly by type inference (3.6), e.g. when invoking arrayInstance.push("foo").

在一个类的定义中使用类型参数时，除非加入了约束（第3.2.1节）否则这些类型参数都没有指定特定的类型。因此编译器必须假定这些类型参数可以被分配为任意类型使用。因此，不能访问类型参数的字段或者 类型转换（第5.23）为一个类型参数类型。也不可能为一个类型参数创建新的实例，除非类型参数是泛型（第3.3节）并且被相应的约束。

Inside the definition of a class with type parameters,these type parameters are an unspecific type. Unless constraints (3.2.1) are added, the compiler has to assume that the type parameters could be used with any type. As a consequence, it is not possible to access fields of type parameters or cast (5.23) to a type parameter type. It is also not possible to create a new instance of a type parameter type, unless the type parameter is generic (3.3) and constrained accordingly.

下面的表格展示了允许声明类型参数的地方：

The following table shows where type parameters are allowed:

位置	触发绑定	注意
Class	实例化	也可以于成员字段被访问时绑定
Enum	实例化	
Enum构造函数	实例化	
Function	调用中	允许用于方法和命名的局部 lvalue 函数
Structure	实例化	

函数类型参数在函数被调用时触发绑定，这样的类型参数（如果无约束）接受任何类型。但是，每次调用只接受一种类型。比如当一个函数有多个参数时：

With function type parameters being bound upon invocation, such a type parameter (if unconstrained) accepts any type. However, only one type per invocation is accepted. This can be utilized if a function has multiple arguments:

```
class Main {
   static public function main() {
       equals(1, 1);
       // runtime message: bar should be foo
       equals("foo", "bar");
       // compiler error: String should be Int
       equals(1, "foo");
   }

   static function equals<T>(expected:T, actual:T) {
       if (actual != expected) {
           trace('$actual should be $expected');
       }
   }
}
```

`equals` 函数的 `expected` 参数和 `actual` 参数都是类型 `T` 。这意味着对于每个 `equals` 的调用，这两个参数必须是相同类型。编译器承认第一个调用（两种参数类型都是 `Int` ）和第二个调用（两个参数都是 `String` ），但是第三个调用引发了一个编译器错误。

Both of the `equals` function's arguments, `expected` and `actual`, have type `T`. This implies that for each invocation of `equals`, the two arguments must be of the same type. The compiler permits the first call (both arguments being of `Int`) and the second call (both arguments being of `String`) but the third attempt causes a compiler error due to a type mismatch.

> 花絮：表达式语法中的类型参数 我们常常被问及，为什么一个使用类型参数的方法不能被以 `method<String>(x)` 形式调用。编译器给出的错误信息不是很有用。这里，有一个简单的解释：上面的代码会把 < 和 > 都解析为二元操作符，结果会解析为 `(method< Sting) > (x)` 。
>
> **Trivia: Type parameters in expression syntax**
>
> We often get the question of why a method with type parameters cannot be called as `method<String>(x)`. The error messages the compiler gives are not very helpful. However, there is a simple reason for that: the above code is parsed as if both `<` and `>` were binary operators, yielding `(method < String) > (x)`.

3.2.1.约束

类型参数可以通过多种类型被约束：

> Type parameters can be constrained to multiple types:

```
typedef Measurable = {
  public var length(default, null):Int;
}

class Main {
  static public function main() {
    trace(test([]));
    trace(test(["bar", "foo"]));
    // String should be Iterable<String>
    // test("foo");
  }

  #if (haxe_ver >= 4)
  static function test<T:Iterable<String> & Measurable>(a:T) {
  #else
  static function test<T:(Iterable<String>, Measurable)>(a:T) {
```

```
  #end
    if (a.length == 0)
      return "empty";
    return a.iterator().next();
  }
}
```

方法 `test` 的类型参数 `T` 被约束为 `Iterable<String>` 和 `Measurable` 。后者为方便起见使用了 typedef（第3.1节）关键字定义且需要兼容有一个 `Int` 类型 名为 `length` 的只读属性（第4.2节）。约束指明一个兼容的类型满足：

> The `test` method contains a type parameter `T` that is constrained to the types `Iterable<String>` and `Measurable`. The latter is defined using a **typedef (3.1)** for convenience and requires compatible types to have a read-only **property (4.2)** named `length` of type `Int`. The constraints then indicate that a type is compatible if:

- 与 `Iterable<String>` 兼容
- 并具有一个 `Int` 类型且名为 `length` 的属性

> - it is compatible with `Iterable<String>` and
> - has a `length` property of type `Int`.

我们可以看到，在第7行调用 `test` 并传递一个空数组，以及第8行传递 `Array<String>` 是没有问题的。因为数组具有 `length` 属性以及 `iterable` 方法。然而，传递一个字符串作为参数，如第9行则会导致约束失败，因为字符串不兼容 `Iterable<T>` 。

> In the above example, we can see that invoking `test` with an empty array on line 7 and an `Array<String>` on line 8 works fine. This is because `Array` has both a `length` property and an `iterator` method. However, passing a `String` as argument on line 9 fails the constraint check because `String` is not compatible with `Iterable<T>`.

3.3.泛型

通常，Haxe编译器只生成一个单独的类或者函数，即使它有类型参数。生成的代码之后可能必须执行一些类型检查可能会影响一些性能。这发生在一个自然抽象概念中，目标语言的代码生成器必须假设一个类型参数可以是任何类型。

> Usually, the Haxe Compiler generates only a single class or function even if it has type parameters. This results in a natural abstraction where the code generator for the target language has to assume that a type parameter could be of any type. The generated code then might have to perform some type checks which can be detrimental to performance.

一个类或者函数可以通过使用 *:generic* 元数据（第*6.9*节）归类为泛型。这导致编译器每个类型参数和破损的名字的混合，发射一个不同的类/函数。一个这样的规范可以使得静态目标语言（第2.2节）的部分 性能关键型 **(performance-critical)** 的代码得到性能提升，但代价是其生成的体量变大：

> A class or function can be made generic by attributing it with the :generic metadata (6.9). This causes the compiler to emit a distinct class/function per type parameter combination with mangled names. A specification like this can yield a boost in performance-critical code portions on static targets (2.2) at the cost of a larger output size:

```
@:generic
class MyValue<T> {
    public var value:T;
    public function new(value:T) {
        this.value = value;
    }
}

class Main {
    static public function main() {
        var a = new MyValue<String>("Hello");
        var b = new MyValue<Int>(42);
    }
}
```

似乎不常见这里的显式类型 MyValue，因为我们通常让类型推断（第3.6节）处理它。尽管如此，它确实需要在这种情况下被需要。编译器必须了解泛型类的准确类型一经构建。JavaScript输出显示结果：

> It seems unusual to see the explicit type MyValue here as we usually let type inference (3.6) deal with this. Nonetheless, it is indeed required in this case. The compiler has to know the exact type of a generic class upon construction. The JavaScript output shows the result:

```
(function () { "use strict";
    var Test = function() { };
    Test.main = function() {
        var a = new MyValue_String("Hello");
        var b = new MyValue_Int(5);
    };
    var MyValue_Int = function(value) {
        this.value = value;
    };
    var MyValue_String = function(value) {
        this.value = value;
    };
    Test.main();
```

```
})();
```

我们可以确定 MyValue和MyVlue 已经变成 MyValue_String 和MyValue_Int 。这类似于泛型函数：

> We can identify that MyValue and MyValue have become MyValue_String and MyValue_Int respectively. This is similar for generic functions:

```
class Main {
    static public function main() {
        method("foo");
        method(1);
    }

    @:generic static function method<T>(t:T) { }
}
```

再一次，JavaScript输出使其非常明显：

> Again, the JavaScript output makes it obvious:

```
(function () { "use strict";
    var Main = function() { }
    Main.method_Int = function(t) {
    }
    Main.method_String = function(t) {
    }
    Main.main = function() {
        Main.method_String("foo");
        Main.method_Int(1);
    }
    Main.main();
})();
```

3.3.1.泛型类型参数解释

> 定义：泛型类型参数 如果它的包含类或者方法是泛型，则一个类型参数被认为是泛型。 [warning] Definition: Generic Type Parameter A type parameter is said to be generic if its containing class or method is generic.

不可能构建一般的类型参数，例如 new T() 会有编译器错误。理由是Haxe只生成一个单独的函数，而且构造毫无意义。当类型参数为泛型的时候则不同：因为我们知道编译器会生成一个不同的函数对于每个类型参数组合，可以使用真实的类型替换 T new T()。

> It is not possible to construct normal type parameters, e.g. new T() is a compiler error. The reason for this is that Haxe generates only a single function and the construct makes no sense in that case. This is different when the type parameter is generic: Since we know that the compiler will generate a distinct function for each type parameter combination,it is possible to replace the T new T() with the real type.

```
typedef Constructible = {
    public function new(s:String):Void;
}

class Main {
    static public function main() {
        var s:String = make();
        var t:haxe.Template = make();
    }
```

```
    @:generic
    static function make<T:Constructible>():T {
        return new T("foo");
    }
}
```

应该注意，这里使用从下到下的推断（第3.6.1节）来确定T的实际类型。这种类型的类型参数构建有两个需求才能工作：构建的类型参数必须是

> It should be noted that top-down inference (3.6.1) is used here to determine the actual type of T. There are two requirements for this kind of type parameter construction to work: The constructed type parameter must be

- 是泛型，而且
- 显式约束（第3.2.1节）为有一个构造参数（第2.3.1节）

> - generic and
> - be explicitly constrained (3.2.1) to having a constructor (2.3.1).

这里，1，通过使用有 @generic 元数据，2. 通过 T 被约束为可构造的。这个约束适用于String 和haxe.Template 因为它们都有一个构造函数，接受一个单一的Sting参数。果然，相关的JavaScript输出看起来和预期的一样：

> Here, 1. is given by make having the @:generic metadata, and 2. by T being constrained to Constructible. The constraint holds for both String and haxe.Template as both have a constructor accepting a singular String argument. Sure enough, the relevant JavaScript output looks as expected:

```
var Main = function() { }
Main.__name__ = true;
Main.make_haxe_Template = function() {
    return new haxe.Template("foo");
}
Main.make_String = function() {
    return new String("foo");
}
Main.main = function() {
    var s = Main.make_String(); 11 var t = Main.make_haxe_Template();
}
```

> **Define: Generic Type Parameter**
>
> A type parameter is said to be generic if its containing class or method is generic.

It is not possible to construct normal type parameters; for example, `new T()` would register as a compiler error. The reason for this is that Haxe generates only a single function and the construct would make no sense in that case. This is different when the type parameter is generic: since we know that the compiler will generate a distinct function for each type parameter combination, it is possible to replace the `T` `new T()` with the real type.

```
import haxe.Constraints;

class Main {
  static public function main() {
    var s:String = make();
    var t:haxe.Template = make();
  }

  @:generic
```

```
  static function make<T:Constructible<String->Void>>():T {
    return new T("foo");
  }
}
```

It should be noted that top-down inference is used here to determine the actual type of `T`. For this kind of type parameter construction to work, the constructed type parameter must meet two requirements:

1. It must be generic.
2. It must be explicitly constrained to have a constructor.

Here, the first requirement is met by `make` having the `@:generic` metadata, and the second by `T` being constrained to `Constructible`. The constraint holds for both `String` and `haxe.Template` as both have a constructor accepting a singular `String` argument. Sure enough, the relevant JavaScript output looks as expected:

```
var Main = function() { }
Main.__name__ = true;
Main.make_haxe_Template = function() {
    return new haxe.Template("foo");
}
Main.make_String = function() {
    return new String("foo");
}
Main.main = function() {
    var s = Main.make_String();
    var t = Main.make_haxe_Template();
}
```

3.4.变异

虽然许多地方可以产生变异，但它通常在伴随类型参数出现时使人感到意外。另外变异错误也非常容易被触发：

> While variance is also relevant in other places, it occurs particularly often with type parameters and comes as a surprise in this context. Additionally, it is very easy to trigger variance errors:

```
class Base {
    public function new() { }
}

class Child extends Base { }

class Main {
    public static function main () {
        var children = [new Child()];
        // Array<Child> should be Array<Base>
        // Type parameters are invariant
        // Child should be Base
        var bases:Array<Base> = children;
    }
}
```

显然，一个 `Array<Child>` 不能被赋值到一个 `Array<Base>` 上，尽管 `Child` 可以被赋值到 `Base` 。原因可能有些出乎意料：由于 array 可以被写入，比如通过它们的 `push()` 方法添加新的元素，因此不允许此类赋值操作。如果忽略变异错误非常容易产生问题：

> Apparently,an `Array` cannot be assigned to an `Array` ,even though `Child` can be assigned to `Base` . The reason for this might be somewhat unexpected: It is not allowed because arrays can be written to, e.g. via their `push()` method. It is easy to generate problems by ignoring variance errors:

```
class Base {
    public function new() { }
}

class Child extends Base { }

class OtherChild extends Base { }

class Main {
    public static function main () {
        var children = [new Child()];
        // subvert type checker
        var bases:Array<Base> = cast children;
        bases.push(new OtherChild());
        for(child in children) {
            trace(child);
        }
    }
}
```

这里我们使用 cast（第5.23节）推翻了类型检查，因此允许了注释行后的赋值操作。此时我们通过 `base` 持有了一个指向原数组的引用，且将其类型化为 `Array<Base>` 。这样做我们就可以为其推入（push）一个可与 `Base` 类型兼容的类型，比如此例中的 `OtherChild` 。然而，由于引用指向的原始数组是 `children` ，其类型为 `Array<Child>` ，如果我们对原始数组的元素进行遍历，会在碰上 `OtherChild` 实例的时候出问题。

> Here we subvert the type checker by using a cast (5.23), thus allowing the assignment after the commented line. With that we hold a reference `bases` to the original array, typed as `Array`. This allows pushing another type compatible with `Base` (OtherChild) onto that array. However,our original reference `children` is still of type `Array` and things go bad when we encounter the `OtherChild` instance in one of its elements while iterating.

如果 `Array` 没有 `push()` 方法，且没有其它修改的手段，赋值就是安全的，因为不会有不兼容的类型被添加进数组中。在 Haxe 中，我们可以通过结构化子类型（第3.5.2节）来对类型进行相应的约束。

> If `Array` had no `push()` method and no other means of modification, the assignment would be safe as no incompatible type could be added to it. This can be achieved by restricting the type accordingly using structural subtyping:

```
class Base {
    public function new() { }
}

class Child extends Base { }

typedef MyArray<T> = {
    public function pop():T;
}

class Main {
    public static function main () {
        var a = [new Child()];
        var b:MyArray<Base> = a;
    }
}
```

这样我们就可以安全地将其赋值给 `MyArray<Base>` 类型的变量 `b` ，因为 `MyArray` 只有一个 `pop()` 方法。`MyArray` 中也没有定义可用于添加不兼容类型的方法。此时称其发生了协变。

> We can safely assign with `b` being typed as `MyArray<Base>` and `MyArray` only having a `pop()` method. There is no method defined on `MyArray` which could be used to add incompatible types. It is thus said to be **covariant**.

协变

> 定义 协变：
>
> 如果一个复合类型的组分的类型可以被赋值到一个不那么明确（less specific）的类型的组分上，该复合类型便被认为发生了协变，也即是说组分只被读取，不被写入。
>
> [warning] Definition Covariance:
>
> A compound type is considered covariant if its component types can be assigned to less specific components, i.e. if they are only read, but never written.
>
> 定义 逆变： 如果一个复合类型的组分的类型可以被赋值到一个更为具体（less generic）的类型的组分上，该复合类型便被认为发生了逆变，也即是说组分只被写入，不被读取。
>
> [warning] Definition Contravariance:
>
> A compound type is considered contravariant if its component types can be assigned to less generic components, i.e. if they are only written, but never read.

3.5.一致性检查

一致性检查 是 Haxe 类型系统的核心，其为Haxe程序极大地提升了健壮性（或称鲁棒性）。它描述了某一个类型与另一个类型是否兼容的过程。

> *Unification is the heart of the type system and contributes immensely to the robustness of Haxe programs. It describes the process of checking if a type is compatible to another type.*
>
> 定义 一致性检查： 类型A 与类型 B 之间的一致性检查是一个定向过程，它会回答一个问题：A 是否可以被赋值给 B。当两者中有一个 是/含有 单形（monomorph) (2.9) 时，其类型将在一致性检查过程中变形（mutate）
>
> [warning] Definition: Unification Unification between two types A and B is a directional process which answers the question if A can be assigned to B. It may mutate either type if it is or has a monomorph (2.9).

一致性错误非常容易被触发：

Unification errors are very easy to trigger:

```
class Main {
   static public function main() {
       // Int should be String
       var s:String = 1;
   }
}
```

当我们尝试分配一个 `Int` 类型的值给一个 `String` 类型变量时，会导致编译器尝试 把 `Int` 统一为 `String` 。当然，这是不允许的并且会使编译器发出一个错误 “Int should be String”。

> *We try to assign a value of type Int to a variable of type String, which causes the compiler to try and unify Int with String. This is, of course, not allowed and makes the compiler emit the error Int should be String.*

在这个特殊的例子中，一致性检查在 赋值 操作时被触发，前文中的 “...是否可以被赋值给...” 的定义在此例中很直观，但这只是会执行一致性检查的其中一例：

> *In this particular case, the unification is triggered by an assignment, a context in which the “is assignable to” definition is intuitive. It is one of several cases where unification is performed:*

赋值：如果 a 被分配给 b，类型 a 会统一为 类型 b 。

> *Assignment: If `a` is assigned to `b`, the type of `a` is unified with the type of `b`.*

函数调用：我们在介绍 函数类型 时已经看过一个简要的例子了。通常，编译器会尝试把第一个实际参数的类型统一为第一个形式参数的类型、把第二个实际参数类型统一为第二个形式参数的类型...以此类推。

> *Function call: We have briefly seen this one while introducing thefunction(2.6)type. In general, the compiler tries to unify the first given argument type with the first expected argument type, the second given argument type with the second expected argument type and so on until all argument types are handled.*

函数返回：不论函数何时出现 `return e` 表达式， `e` 的类型都会被统一为函数的返回类型。如果函数没有显式地声明返回类型，则返回类型会被推断为 `e` 的类型 且 随后的 `return` 表达式都会以该类型进行推断。

> *Function return: Whenever a function hasa return e expression, thetypeof e is unified with the function return type. If the function has no explicit return type,it is inferred to the type of e and subsequent `return` expressions are inferred against it.*

数组声明：编译器会尝试在一个数组声明的所有给定的类型中寻找一个最小化类型。请参阅 通用基本类型（第3.5.5节）了解更多细节。

> *Array declaration: The compiler tries to find a minimal type between all given types in an array declaration. Refer to Common Base Type (Section 3.5.5) for details.*

对象声明：如果一个对象被分配了一个与声明时所不同的类型，那么编译器会把所有给定类型的字段的类型统一为期望类型的字段的类型。

> *Object declaration: If an object is declared "against" a given type, the compiler unifies each given field type with each expected field type.*

操作符一致性检查：某一操作符会期望作用于某一种类型上，当操作符作用于某一类型时该类型会被统一为操作符期望的类型，比如，表达式 `a && b` 会把 `a` 和 `b` 都统一为 `Bool` 类型， 而表达式 `a == b` 会把 `a` 统一为 `b`。

> *Operator unification: Certain operators expect certain types which the given types are unified against. For instance, the expression a && b unifies both a and b with Bool and the expression a == b unifies a with b.*

3.5.1.类与接口

当定义类之间的一致性检查行为时需要特别注意，一致性是定向检查的：也就是说我们可以赋值一个特例化的类型（例如一个子类）到一个通用类型（例如一个父类），但是反过来是不允许的。

> *When defining unification behavior between classes,it is important to remember that unification is directional: We can assign a more specialized class (e.g. a child class) to a generic class (e.g. a parent class) but the reverse is not valid.*

如下的赋值是允许的：

- 子类赋值给父类
- 类赋值给其所实现的接口
- 接口赋值到基接口

> *The following assignments are allowed:*
>
> - *child class to parent class*
> - *class to implementing interface*
> - *interface to base interface*

这组规则是可传递的，这意味着一个子类也可以被赋值到其基类的基类、其基类所实现的接口、实现的接口的基接口等，以此类推。

> *These rules are transitive, meaning that a child class can also be assigned to the base class of its base class,an interface its base class implements,the base interface of an implementing interface and so on.*

3.5.2.结构子类型

> 定义 结构子类型化： 结构子类型化定义了具有相同结构的类型之间的一种隐式关系。
>
> [warning] Definition: Structural Subtyping Structural subtyping defines an implicit relation between types that have the same structure.

Haxe中当以下情况发生时允许结构子类型化：

- 一个类（第2.3节）统一为 一个结构（第2.5节）时，
- 一个结构 统一为 另外一个结构时

> *Structural sub-typing in Haxe is allowed when unifying:*
>
> - *a class (2.3) with a structure (2.5) and*
> - *a structure with another structure.*

以下的例子是 Haxe 标准库中 `Lambda` 类的一部分：

> *The following example is part of the Lambda class of the Haxe Standard Library :*

```
public static function empty<T>(it : Iterable<T>):Bool {
    return !it.iterator().hasNext();
}
```

`empty` 方法用于检查一个 `Iterable` （可迭代的）是否含有元素。为了达到这个目的，我们只需要知道参数是一个 `Iterable` 而不需要知道该参数的具体类型。这使得我们可以传递一个可被统一为 `Iterable<T>` 的任意类型的参数来调用 `empty` 方法，Haxe 标准库中的许多类型都满足这一要求。

> *The empty-method checks if an Iterable has an element. For this purpose, it is not necessary to know anything about the argument type other than the fact that it is considered an iterable. This allows calling the empty-method with any type that unifies with Iterable which applies to a lot of types in the Haxe Standard Library.*

尽管这样的类型化非常方便，但是大量地使用可能影响静态目标语言的性能，在 性能影响（第2.5.4节）部分有介绍。

> *This kind of typing can be very convenient but extensive use may be detrimental to performance on static targets, which is detailed in Impact on Performance (Section 2.5.4).*

3.5.3.单形

一个 是/含有 单形（第2.9节）的类型的一致性检查行为在 类型推断（第3.6节）中详述。

> *Unification of types having or being a monomorph(2.9) is detailed in Type Inference(Section3.6).*

3.5.4.函数返回

函数返回类型的一致性检查可能涉及 `Void` 类型（第2.1.5），因此需要对何种类型可以被统一为 `Void` 类型给出一个明确的定义。由于 `Void` 用于描述一个类型的缺省，因此它不能被分配到其他任何类型上，即使是 `Dynamic` 类型也不行 。这意味着如果一个函数显式声明其返回类型为 `Dynamic` ，它便不能返回 `Void` 。

> *Unification of function return types may involve the Void-type (2.1.5) and requires a clear definition of what unifies with Void. With Void describing the absence of a type,it is not assignable to any other type, not even Dynamic. This means that if a function is explicitly declared as returning Dynamic, it cannot return Void.*

反之也一样：如果一个函数声明其返回类型为 `Void` ，那么它便不能返回 `Dynamic` 或者其他任何类型。但是这个方向上的一致性检查在函数类型的赋值操作上是被允许的：

> *The opposite applies as well: If a function declares a return type of Void, it cannot return Dynamic or any other type. However, this direction of unification is allowed when assigning function types:*

```
var func:Void->Void = function() return "foo";
```

rvalue 函数显然是 `Void->String` 类型的，但我们可以把它分配给 `Void->Void` 类型的变量 `func` 。这是因为编译器此时可以安全地假定 `func` 的返回类型是无关的，鉴于它不能被分配给任何 非`Void` 类型。

> *The right-hand function clearly is of type Void->String,yet we can assign it to the variable func of type Void->Void. This is because the compiler can safely assume that the return type is irrelevant, given that it could not be assigned to any non-Void type.*

补充：

```
var func:Void -> Void = function() return 'str';
$type(func);//此时 func 的类型依然是 Void -> Void ，因此调用表达式 func() 会得到 Void 返回
```

3.5.5.通用基本类型

给定一个由多个类型构成的集合，若该集合中所有类型都能被统一至某一类型，则称该类型即为该集合的通用基本类型：

> *Given a set of multiple types,a common base type is a type which all types of the set unify against:*

```
class Base {
    public function new() { }
}

class Child1 extends Base { }
class Child2 extends Base { }

class Main {
    static public function main() {
        var a = [new Child1(), new Child2()];
        $type(a); // Array<Base>
    }
}
```

虽然 `Base` 没有被提及，Haxe编译器也能推断它为 `Chid1` 和 `Child2` 的通用类型 。Haxe编译器在以下情况下执行此类一致性检查：

> *Although Base is not mentioned, the Haxe Compiler manages to infer it as the common type of Child1 and Child2. The Haxe Compiler employs this kind of unification in the following situations:*

- 数组声明
- `if` / `else`
- `switch` 的各个 case 分支中

3.6.类型推断

类型推断的效果将会频繁出现于整个文档中。一个例子可以展示类型推断：

> *The effects of type inference have been seen throughout this document and will continue to be important. A simple example shows type inference at work:*

```
class Main {
    public static function main() {
        var x = null;
        $type(x); // Unknown<0>
        x = "foo";
        $type(x); // String
    }
}
```

这里有个特殊的 `$type` 指令在之前为了便于 函数类型（2.6） 的说明稍有提及，现在让我们来正式地介绍它：

> *The special construct $type was previously mentioned in order to simplify the explanation of the Function Type (Section 2.6) type, so let us now introduce it officially:*
>
> 定义 $type： $type 是一个可以像函数一样调用的编译时机制，它接受一个参数。编译器执行参数的表达式并输出表达式的类型。
>
> [warning] Construct: $type *$type is a compile-time mechanism being called like a function,with a single argument. The compiler evaluates the argument expression and then outputs the type of that expression.*

在上面的例子中，第一个 `$type` 输出 `Unknow<0>` 。这是一个 单形（第2.9节），一个还不知道其类型的类型。下一行 `x = "foo"` 将一个 `String` 类型的字面值赋值到变量 `x` 上，于是触发了这个单形与 `String` 之间的一致性检查 。然后我们可以看到 `x` 的类型被改变为 `String` 类型了 。

> *In the example above, the first `$type` prints `Unknown<0>`. This is a monomorph, a type that is not yet known. The next line `x = "foo"` assigns a `String` literal to `x`, which causes the unification of the monomorph with `String`. We then see that the type of `x` has changed to `String`.*

每当一个不同于 Dynamic（第2.7节）的类型被统一为一个单形时，这个单形便会变形（morph）为该类型，并且在此之后它就不能再变形为另外的类型了。这一特性正如其名字中的 mono（单一的）所表达的一样。

> *Whenever a type other than Dynamic is unified with a monomorph, that monomorph **morphs** into that type, or in simpler terms, **becomes** that type. Therefore, it cannot morph into a different type afterwards, a property expressed in the **mono** part of its name.*

遵循一致性检查的规则，类型推断可以在复合类型中触发：

> *Following the rules of unification, type inference can occur in compound types:*

```
class Main {
    public static function main() {
        var x = [];
        $type(x); // Array<Unknown<0>>
        x.push("foo");
        $type(x); // Array<String>
    }
}
```

变量 `x` 在一开始被初始化为一个空的 `Array` 数组。此时我们除了知道变量 `x` 是一个数组外并不知道其数组元素的具体类型。此时 `x` 的类型必然为 `Array<Unknown<0>>` 。只有当我们为其推入一个 `Sring` 类型元素之后才知道其类型为 `Array<String>` 。

> *Variable `x` is first initialized to an empty `Array`. At this point, we can tell that the type of `x` is an array, but we do not yet know the type of the array elements. Consequently, the type of `x` is `Array<Unknown<0>>`. It is only after pushing a `String` onto the array that we know the type to be `Array<String>`.*

3.6.1.由上而下推断

大多数时候，类型会被自行推断后可能被统一为一个预期类型。然而在一些地方一个预期类型可能会被用来影响推断，此时我们称其为自上而下的推断。

> *Most of the time,types are inferred on their own and may then be unified with an expected type. In a few places,however,an expected type may be used to influence inference. We then speak of top-down inference.*
>
> 定义 预期类型： 当表达式被类型化之前，表达式所处位置的类型是已知的，称该类型为预期类型，例如，当一个表达式作为一个函数调用的参数传递时，参数类型可以通过所谓的自上而下推断来影响表达式的类型化过程。
>
> [warning] Definition: Expected Type Expected types occur when the type of an expression is known before that expression has been typed, e.g. because the expression is argument to a function call. They can influence typing of that expression through what is called top-down inference (3.6.1).

一个很好的例子是混合类型的数组。如在 Dynamic（第2.7节）中提到的，编译器不允许 [1,"foo"] ，因为编译器无法确定元素的类型。但是通过自上而下的推断可以绕过这个问题：

> *A good example are arrays of mixed types. As mentioned in Dynamic (Section 2.7), the compiler refuses [1, "foo"] because it cannot determine an element type. Employing top-down inference, this can be overcome:*

```
class Main {
   static public function main() {
       var a:Array<Dynamic> = [1, "foo"];
   }
}
```

这里，编译器知道类型化 [1,"foo"] 时的预期类型是 `Array<Dynamic>` ，因此元素类型为 `Dynimic` ，不同于常规的一致性检查行为下编译器会尝试（此例会失败）去寻找一个通用基本类型（第3.5.5节），此时每一个元素都会被统一至并类型化为 `Dynamic` 类型。

> *Here, the compiler knows while typing `[1, "foo"]` that the expected type is `Array<Dynamic>`, so the element type is `Dynamic`. Instead of the usual unification behavior where the compiler would attempt (and fail) to determine a common base type, the individual elements are typed against and unified with `Dynamic`.*

在介绍构造泛型类型参数（第3.3.1节）时，我们看到了另一个自上而下推断的有趣用法

> We have seen another interesting use of top-down inference when construction of generic type parameters (3.3.1) was introduced:

```
import haxe.Constraints;

class Main {
```

```
    static public function main() {
      var s:String = make();
      var t:haxe.Template = make();
    }

    @:generic
    static function make<T:Constructible<String->Void>>():T {
      return new T("foo");
    }
  }
```

显式的类型声明 `String` 与 `haxe.Template` 在此处被用于确定 `make` 的返回类型。能这么做是因为我们通过 `make()` 调用该方法，所以知道返回类型会被赋值到变量上。利用这个信息，我们可以把 未知类型（Unknown Type） `T` 分别绑定至 `String` 和 `haxe.Template` 类型上 。

> *The explicit types `String` and `haxe.Template` are used here to determine the return type of `make`. This works because the method is invoked as `make()`, so we know the return type will be assigned to the variables. Utilizing this information, it is possible to bind the unknown type `T` to `String` and `haxe.Template` respectively.*

3.6.2.局限

类型推断在使用局部变量的时候节省了大量手动类型化，但是有时类型系统仍然需要一些帮助。事实上，它甚至不尝试推断变量（第4.1节）或者属性（第4.2节）字段，除非它有一个直接的初始化。

> Type inference saves a lot of manual type hints when working with local variables,but sometimes the type system still needs some help. In fact, it does not even try to infer the type of a variable (4.1) or property (4.2) field unless it has a direct initialization.

也有一些情况递归调用，当类型推断有限制的时候。如果一个函数递归调用它自身，它的类型还不（完全）不知道，类型推断可能推断一个错误，太详细的类型。

> There are also some cases involving recursion where type inference has limitations. If a function calls itself recursively while its type is not (completely) known yet, type inference may infer a wrong, too specialized type.

一种不同类型的限制涉及代码的可读性。如果类型推断过度使用，可能由于可见类型的缺失而难于理解程序的部分。特别是在方法签名。建议在类型推断和显式类型中找到一个好的平衡。

> A different kind of limitation involves the readability of code. If type inference is overused it might be difficult to understand parts of a program due to the lack of visible types. This is particularly true for method signatures. It is recommended to find a good balance between type inference and explicit type hints.

3.7.模块和路径

> 定义：模块 所有的Haxe代码组织在模块中，通过路径定位。本质上讲，每个.hx 文件表示一个模块，其中可能包含多个类型，其中的类型可能是修饰为 `private` 的私有类，这种情况下只有在它所处的模块之内才能访问到它。 [warning] Definition: Module All Haxe code is organized in modules, which are addressed using paths. In essence, each .hx file represents a module which may contain several types. A type may be private, in which case only its containing module can access it.

模块和它所包含的同名类型之间的区别在设计上是模糊的。事实上，定位 `haxe.ds.StringMap<Int>` 可以认为是 `haxe.ds.StringMap.StringMap<Int>` 的简短写法。后者由四部分构成：

The distinction between a module and its containing type of the same name is blurry by design. In fact, addressing `haxe.ds.StringMap<Int>` can be considered shorthand for `haxe.ds.StringMap.StringMap<Int>`. The latter version consists of four parts:

1. 包名 `haxe.ds`
2. 模块名 `StringMap`
3. 类型名 `StringMap`
4. 类型参数 `Int`

如果模块和类型名是相同的，重复的部分可以被省略，使用 `haxe.ds.StringMap<Int>` 作为速记。然而，了解扩展的记法可以帮助我们理解 模块子类型（3.7.1）是如何定位的。

If the module and type name are equal, the duplicate can be removed, leading to the `haxe.ds.StringMap<Int>` short version. However, knowing about the extended version helps with understanding how module sub-types are addressed.

路径可以通过使用 import 进一步简化，这么做通常可以省略路径中包名的部分，其中涉及无约束标识符的使用，

Paths can be shortened further by using an import(3.7.2), which typically allows omitting the package part of a path. This may lead to usage of unqualified identifiers, for which understanding the resolution order (3.7.3) is required.

> 定义：类型路径 类型的点路径由包，模块名和类型名组成。它的通常格式是 pack1.paceN.ModuleName.TypeName 。 [warning] Definition: Type path The (dot-)path to a type consists of the package, the module name and the type name. Its general form is pack1.pack2.packN.ModuleName.TypeName.

3.7.1.模块子类型

模块子类型是一个模块中与模块不同名的类型。这使得一个 .hx 文件中可以包含多个类型声明，这些类型可以在该模块中不受限制地被调用，但从其它模块中使用时需通过 `package.Modoule.Type` 的形式进行访问:

A module sub-type is a type declared in a module with a different name than that module. This allows a single .hx file to contain multiple types, which can be accessed unqualified from within the module, and by using package.Module.Type from other modules:

```
var e:haxe.macro.Expr.ExprDef;
```

此处访问了 `haxe.macro.Expr` 模块中的模块子类型 `ExprDef` 。

模块子类型的声明可能看起来如下所示：

```
// a/A.hx
package a;

class A { public function new() {} }
// sub-type
class B { public function new() {} }
```

```
// Main.hx
import a.A;
class Main {
    static function main() {
        var subtype1 = new a.A.B();

        // these are also valid, but require import a.A or import a.A.B
        var subtype2 = new B();
        var subtype3 = new a.B();
    }
}
```

模块子类型的关系不会反映于运行时；一个修饰为 public 的模块子类型会被认为是其所在的包（package）的一个成员，比如在上面的例子中，模块子类型 `ExprDef` 会被生成为 `haxe.macro.ExprDef`。所以当一个包中存在两个模块，且两个模块中定义了同名的模块子类型时可能会产生冲突。通常，Haxe编译器会侦测到这类情况并做出相应报告。

模块子类型也可以被声明为私有的（private）：

```
private class C {...}
private enum E {...}
private typedef T {...}
private abstract A {...}
```

> 定义：私有类型 一个类型可以通过 `private` 访问修饰符被声明为私有。这会导致这个类型只能从定义它的这个模块内部直接进行访问。私有类型不像公开类型，不会成为它们所在的包的成员。 *Definition: Private type A type can be made private by using the private modifier. As a result,the type can only be directly accessed from within the module (3.7) it is defined in. Private types, unlike public ones, do not become a member of their containing package.*

类型的可访问性可以通过访问控制（第6.10节）进行更精确地控制。

The accessibility of types can be controlled more precisely by using access control.

3.7.2.Import

如果一个类型路径在一个 .hx 文件中被多次使用，那么最好是通过导入该路径来缩短书写形式。这么做使得我们使用一个类型时可以省略其包名：

> If a type path is used multiple times in a .hx file,it might make sense to use an import to shorten it. This allows omitting the package when using the type:

```
import haxe.ds.StringMap;

class Main {
    static public function main() {
```

```
        // instead of: new haxe.ds.StringMap();
        new StringMap();
    }
}
```

随着haxe.ds.StringMap被导入之后，编译器就可以通过这个包解析 main 函数中不合法的标识符 StringMap。模块 StringMap 被告知需要被导入到当前的文件 。

> With haxe.ds.StringMap being imported in the first line,the compiler is able to resolve the unqualified identifier StringMap in the main function to this package. The module StringMap is said to be imported into the current file.

在这个例子中，我们实际上导入了一个模块，而不只是模块中的一个类型。这意味着所有在被导入的模块中定义的类型都是可用的。

> In this example, we are actually importing a module, not just a specific type within that module. This means that all types defined within the imported module are available:

```
import haxe.macro.Expr;

class Main {
    static public function main() {
        var e:Binop = OpAdd;
    }
}
```

类型 Binop 是haxe.macro.Expr模块中一个 enum（第2.4节）声明，因此在这个模块被导入之后就可以使用了。如果我们只要导入模块中的一个特定类型，例如，import haxe.macro.Expr.ExprDef，程序会编译失败，提示Binop类没有找到。

> The type Binop is an enum (2.4) declared in the module haxe.macro.Expr,and thus available after the import of said module. If we were to import only a specific type of that module, e.g. import haxe.macro.Expr.ExprDef, the program would fail to compile with Class not found : Binop.

关于导入有几个方面需要了解：

> There are several aspects worth knowing about importing:

- 最底部的导入指令具有最高的优先级（详见 解析顺序（第3.7.3节））。
- 静态扩展（第6.3节）关键字 using 具有 import 的效果。
- 如果一个enum被导入（直接或作为模块的一部分被导入），所有它的enum构造函数（第2.4.1节）同样也被导入（这就是为什么在上面例子中允许 opAdd 的用法）。

> - The bottommost import takes priority (detailed in Resolution Order (Section 3.7.3)).
> - The static extension (6.3) keyword using implies the effect of import.
> - If an enum is imported (directly or as part of a module import), all its enum constructors (2.4.1) are also imported (this is what allows the OpAdd usage in the above example).

此外，也可以导入类的静态字段并不受限制的使用它们。

> Furthermore, it is also possible to import static fields (4) of a class and use them unqualified:

```
import Math.random;

class Main {
    static public function main() {
        random();
```

```
    }
}
```

必须特别注意，字段名或者局部变量名和包名的冲突：因为它们优先级高于包，如果有一个名为haxe的局部变量，它们会会阻挡整个haxe包的使用。

> Special care has to be taken with field names or local variable names that conflict with a package name: Since they take priority over packages, a local variable named haxe blocks off usage the entire haxe package.

通配符导入：Haxe允许使用 .* 使 import可以导入一个包中所有的模块、模块中的所有类型或者类型中的所有静态字段。通过以下例子重点了解此类导入操作只能覆盖同级的访问（only crosses a single level）：

> **Wildcard import** Haxe allows using .* to allow import of all modules in a package, all types in a module or all static fields in a type. It is important to understand that this kind of import only crosses a single level as we can see in the following example:

```
import haxe.macro.*;

class Main {
    static function main() {
        var expr:Expr = null;
        //var expr:ExprDef = null; // Class not found : ExprDef
    }
}
```

使用通配符到haxe.macro的导入，使这个包中的 Expr 模块可以被访问，但是它不能使 **Expr** 模块的子类型 **ExprDef** 被访问。这个规则当一个模块被导入时也扩展到静态字段。

> Using the wildcard import on haxe.macro allows accessing Expr which is a module in this package, but it does not allow accessing ExprDef which is a sub-type of the Expr module. This rule extends to static fields when a module is imported.

当使用通配符导入一个包，编译器并不会立即处理包中的所有模块。这意味着这些模块除非被明确使用否则不会被作为输出的一部分生成。

> When using wildcard imports on a package, the compiler does not eagerly process all modules in that package; modules that have not been used explicitly are not part of the generated output.

使用别名导入 如果一个类型或静态字段在一个导入它的模块中经常使用，可以为它引入别名为一个简短的名字。这也可以用来通过给定一个唯一的标识符来消除命名冲突。

> **Import with alias** If a type or static field is used a lot in an importing module it might help to alias it to a shorter name. This can also be used to disambiguate conflicting names by giving them a unique identifier.

```
import String.fromCharCode in f;

class Main {
    static function main() {
        var c1 = f(65);
        var c2 = f(66);
        trace(c1 + c2); // AB
    }
}
```

这里我们导入String.fromCharCode为 f，使我们可以使用 f(65) 和 f(66)。达到和局部变量一样的使用，这个方法是编译时功能，不会有运行时开销。

> Here we import String.fromCharCode as f which allows us to use f(65) and f(66). While the same could be achieved with a local variable, this method is compile-time exclusive and guaranteed to have no run-time overhead.

从**Haxe3.2.0**后，**Haxe**允许使用更自然的 **as** 替代 **in** 。

> **Since Haxe 3.2.0** Haxe also allows the more natural as in place of in.

3.7.3.解析顺序

解析顺序一被引入就涉及到不受限制的标识符 。如 foo()，foo=1，foo.field等这些是表达式（第5章）。特别是最后一个包括类似 haxe.ds.StringMap的模块路径，haxe是一个绝对的标识符。

> Resolution order comes into play as soon as unqualified identifiers are involved. These are expressions(5)intheformoffoo(),foo = 1andfoo.field. The last one in particular includes module paths such as haxe.ds.StringMap, where haxe is an unqualified identifier.

我们描述解析顺序的算法，取决于以下的状态：

> We describe the resolution order algorithm here, which depends on the following state:

- 声明的局部变量（第5.10节）（包括函数参数）
- 导入（第3.7.2节）的模块，类型和静态字段
- 可用的静态扩展（第6.3节）
- 当前字段的种类（static或者成员）
- 当前类及其父类声明的成员字段
- 当前类声明的静态字段
- 预期的类型（第3.6.1节）
- untyped或者不是untyped的表达式

> - the declared local variables (5.10) (including function arguments)
> - the imported (3.7.2) modules, types and statics
> - the available static extensions (6.3)
> - the kind (static or member) of the current field
> - the declared member fields on the current class and its parent classes
> - the declared static fields on the current class
> - the expected type (3.6.1)
> - the expression being untyped or not

给定一个标识符 i ，算法规则如下：

> Given an identifier i, the algorithm is as follows:

1. 如果i 是true，false，this，super或者null，解析到匹配的 常量并停止
2. 如果一个局部变量命名为 i为可访问的，解析它并停止
3. 如果当前的字段是静态的，跳到6
4. 如果当前的类或者任何它的父类有一个字段命名为 i，解析到它并停止
5. 如果一个静态扩展带有第一个当前类类型的参数可用，解析到它并停止
6. 如果当前的类有一个静态字段命名为 i，解析到它并停止
7. 如果一个枚举构造函数命名为i声明在一个导入的枚举，解析到它并停止

8. 如果一个静态方法 i 被显式导入，解析到它并停止
9. 如果i通过一个小写字母开始，跳到11
10. 如果类型名为 i 是可用的，解析到它并停止
11. 如果表达式不是 untyped模式，跳到14
12. 如果i和这个相等，解析到这个常量并停止
13. 产生一个局部变量命名为 i，解析到它并停止
14. 失败

> 1. If i is true, false, this, super or null, resolve to the matching constant and halt.
> 2. If a local variable named i is accessible, resolve to it and halt.
> 3. If the current field is static, go to 6.
> 4. If the current class or any of its parent classes has a field named i, resolve to it and halt.
> 5. If a static extension with a first argument of the type of the current class is available,resolve to it and halt.
> 6. If the current class has a static field named i, resolve to it and halt.
> 7. If an enum constructor named i is declared on an imported enum, resolve to it and halt.
> 8. If a static named i is explicitly imported, resolve to it and halt.
> 9. If i starts with a lower-case character, go to 11.
> 10. If a type named i is available, resolve to it and halt.
> 11. If the expression is not in untyped mode, go to 14
> 12. If i equals **this**, resolve to the this constant and halt.
> 13. Generate a local variable named i, resolve to it and halt.
> 14. Fail

对于第10步，也需要定义类型的解析顺序：

> For step 10, it is also necessary to define the resolution order of types:

1. 如果一个类型名为 i 被导入（直接或者作为模块的部分导入），解析到它并停止
2. 如果当前的包包含一个模块名为 i和类型名为i，解析到它并停止
3. 如果一个类型名为i在顶层可用，解析到它并停止
4. 失败

> 1. If a type named i is imported (directly or as part of a module), resolve to it and halt.
> 2. If the current package contains a module named i with a type named i, resolve to it and halt.
> 3. If a type named i is available at top-level, resolve to it and halt.
> 4. Fail

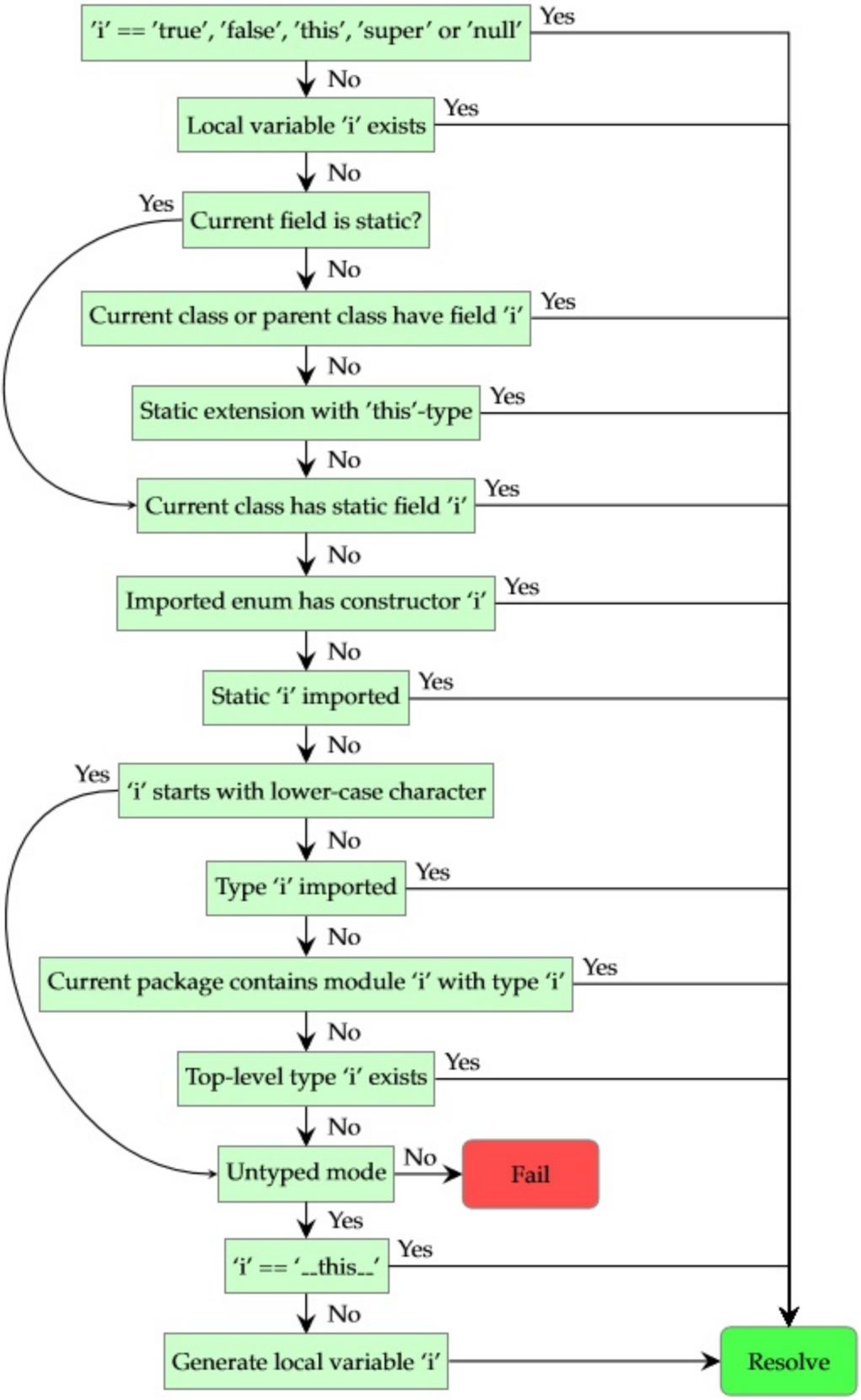
对于这个算法的第一步，和之前的第5部第7步，导入决议的顺序很重要：

> For step 1 of this algorithm as well as steps 5 and 7 of the previous one, the order of import resolution is important:

- 导入的模块和静态扩展从底部到头部检查，第一个匹配的被采用
- 在一个给定模块，类型从头至尾检查
- 对于导入，如果名字相同则为匹配
- 对于静态扩展（第6.3节），如果名字相同并且第一个参数统一（第3.5节）则为匹配。在一个给定类型中被用作静态扩展的字段从头至尾进行检查。

> - Imported modules and static extensions are checked from bottom to top with the first match being picked.
> - Within a given module, types are checked from top to bottom.
> - For imports, a match is made if the name equals.

- For static extensions (6.3), a match is made if the name equals and the first argument unifies (3.5). Within a given type being used as static extension, the fields are checked from top to bottom.



4.类字段

> 定义 类字段： 一个类字段是类中一个静态或非静态的变量、属性、或方法。非静态字段被称为成员字段。比如，我们会说 “一个静态方法” 或 “一个成员变量”。 [warning] Definition: Class Field A class field is a variable, property or method of a class which can either be static or nonstatic. Non-static fields are referred to as member fields, so we speak of e.g. a static method or a member variable.

在前面我们已经了解了 Haxe 程序中的类型是如何构成的，在类字段这一章中我们将结束 Haxe 的 “结构” 部分，并引申至其 “行为” 的部分，这是因为类字段即是 表达式 的归宿。

> So far we have seen how types and Haxe programs, in general, are structured. This section about class fields concludes the structural part and at the same time bridges to the behavioral part of Haxe. This is because class fields are the place where expressions are at home.

有三种类字段：

> There are three kinds of class fields:

- 变量：变量（第4.1节）字段用于保存某种类型的值，可以被读取或者写入。
- 属性：属性（第4.2节）字段对从类之外的访问行为自定了一组访问规则，他们看起来像是一个变量字段。
- 方法：方法（第4.3节）是一个函数，可以被调用以执行一些代码。

> - Variable: A variable(4.1) class field holds a value of a certain type,which can be read or written.
> - Property: A property (4.2) class field defines a custom access behavior for something that, outside the class, looks like a variable field.
> - Method: A method (4.3) is a function which can be called to execute code.

严格说，一个变量可以被认为是一个属性带有某个访问修饰符。事实上，Haxe编译器在它们的类型解析时并不区分变量和属性，但是它们仍然在语法层面上有所区分。

> Strictly speaking, a variable could be considered to be a property with certain access modifiers. Indeed,the Haxe Compiler does not distinguish variables and properties during its typing phase, but they remain separated at syntax level.

术语层面上讲，方法是归属于一个类的（静态或非静态）函数。而诸如出现于表达式中一个局部函数（第5.11节）则不被认为是方法。

> Regarding terminology,a method is a (static or non-static) function belonging to a class. Other functions, such as a local functions (5.11) in expressions, are not considered methods.

4.1.变量

我们在前面的章节中已经见过不少变量字段了，和大部分（不是所有）属性字段一样，变量字段用于存储一个值：

> We have already seen variable fields in several code examples of previous sections. Variable fields hold values, a characteristic which they share with most (but not all) properties:

```
class Main {
    static var member:String = "bar";

    public static function main() {
        trace(member);
        member = "foo";
        trace(member);
    }
}
```

我们可以从这段代码了解到变量具有：

> We can learn from this that a variable

1. 名称（此处为：`member`）
2. 类型（此处为：`String`）
3. 可能带有一个常量初始化值（此处为：`"bar"`）
4. 可能带有访问修饰符（第4.4节）（此处为：`static`）

> 1. has a name (here: member),
> 2. has a type (here: String),
> 3. may have a constant initialization (here: "bar") and
> 4. may have access modifiers (4.4) (here: static)

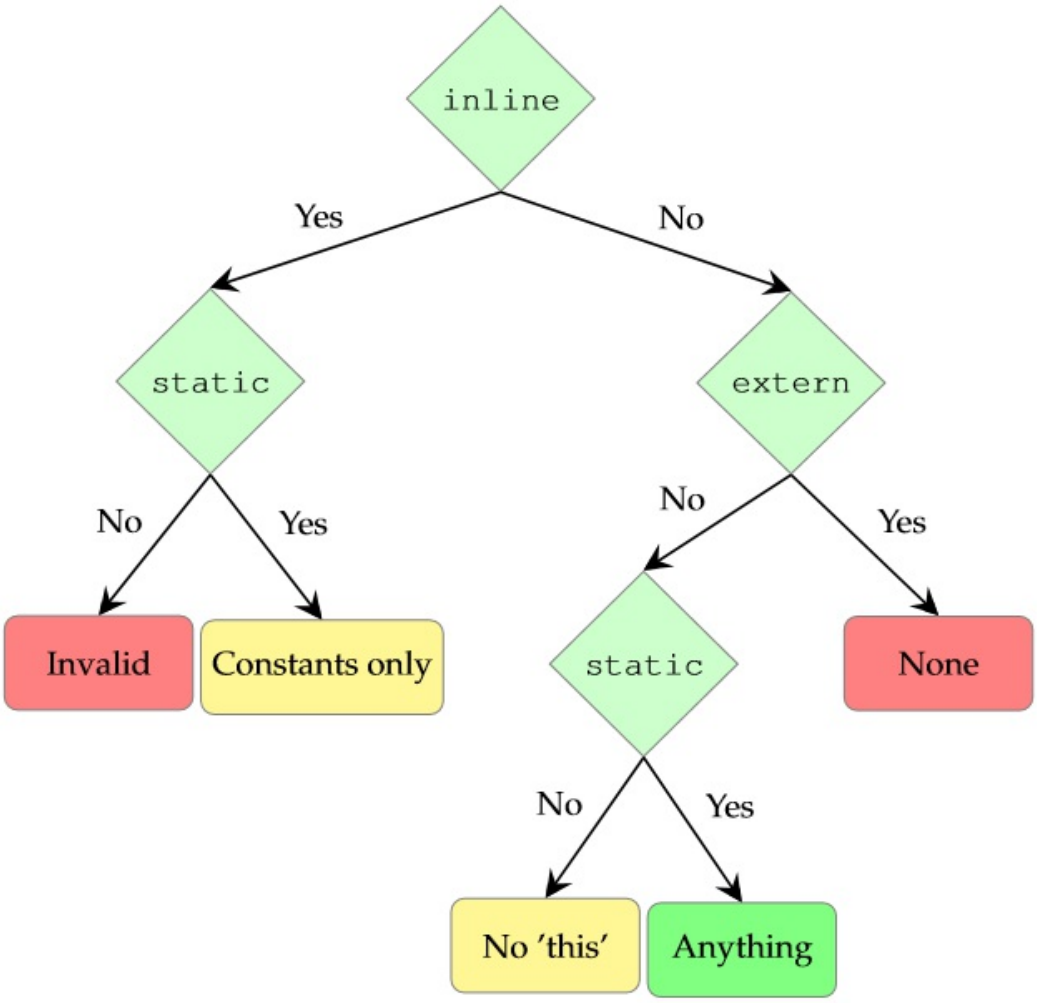
样例程序首先会输出 `menber` 的初始值，然后在输出新的值之前给它赋值 `"foo"`， 访问修饰符的效果由三种类型的类字段共享，将在之后的章节单独展开讨论。

> The example first prints the initialization value of member,then sets it to "foo" before printing its new value. The effect of access modifiers is shared by all three class field kinds and explained in a separate section.

需要注意，如果声明存在初始化值时显式的类型声明不是必须的，此时编译器会自行推断（第 3.6 节）它的类型。

> It should be noted that the explicit type is not required if there is an initialization value. The compiler will infer it in this case.

不同情况下变量字段允许的初始化值：


inline
Yes
No
static
extern
No
Yes
No
Yes
Invalid
Constants only
static
None
No
Yes
No 'this'
Anything

4.2.属性

在变量字段之后，属性字段是第二个用来处理类中的数据的选项。不像变量字段，属性字段能够对字段的访问规则和生成规则提供更细微的控制，通常有以下使用场景：

> Next to variables (4.1), properties are the second option for dealing with data on a class. Unlike variables however, they offer more control of which kind of field access should be allowed and how it should be generated. Common use cases include:

- 一个可以从任何地方读取，但是只能从定义所在的类内部进行写入的字段
- 一个通过 getter 方法进行读取访问的字段
- 一个通过 setter 方法进行写入访问的字段

> - Have a field which can be read from anywhere,but only be written from within the defining class.
> - Have a field which invokes a getter-method upon read-access.
> - Have a field which invokes a setter-method upon write-access.

面对属性字段时，重要的是理解以下两种类型的访问：

> When dealing with properties, it is important to understand the two kinds of access:
>
> 定义 读取访问： 一个字段的读取访问发生于 字段访问表达式（第5.7节）作为右值（right-hand side) 表达式使用时。其中也包括 `obj.field()` 形式的调用，字段 `field` 此时发生了读取访问。
>
> 定义 写入访问： 一个字段的写入访问发生于字段访问表达式被赋值时，比如 `obj.field = value` 。当使用 `+=` 等特殊的赋值操作符时，读取访问也会同时发生，比如表达式 `obj.field += value` 。
>
> [warning] Definition: Read Access A read access to a field occurs when a right-hand side field access expression (5.7) is used. This includes calls in the form of obj.field(), where field is accessed to be read.
>
> Definition: Write Access A write access to a field occurs when a field access expression (5.7) is assigned a value in the form of obj.field = value. It may also occur in combination with read access (4.2) for special assignment operators such as += in expressions like obj.field += value.

读取访问和写入访问行为直接反映在其声明的语法形式上，如下面的例子：

> Read access and write access are directly reflected in the syntax, as the following example shows:

```
class Main {
    public var x(default, null):Int;
    static public function main() { }
}
```

声明的语法大体上和变量字段的语法相似，事实上他们的语法规则也是一样的。但是属性字段的声明有以下不同：

> For the most part, the syntax is similar to variable syntax, and the same rules indeed apply. Properties are identified by

- 字段名称后跟开口括号 `(`
- 后跟一个特定的访问标识符（此例为： `default` ），
- 一个逗号 `,` 进行分隔
- 后跟另一个特殊的访问标识符（此例为： `null` ），
- 最后闭口括号 `)` 进行闭合

> - the opening parenthesis (after the field name,
> - followed by a special access identifier (here: default),
> - with a comma , separating
> - another special access identifier (here: null)
> - before a closing parenthesis).

访问标识符定义了字段的读取（第一个标识符）行为，和写入（第二个标识符）行为。有以下标识符可用：

> The access identifiers define the behavior when the field is read (first identifier) and written (second identifier). The accepted values are:

- `default`：如果字段修饰为 `public` 则具备通常的访问权限，否则等于 `null` 标识符。
- `null`：只允许从定义的类内部进行访问。
- `get` / `set`：访问行为被生成为一个存取器方法进行调用。编译器会确保存取器可用。
- `dynamic`：作用与 `get` / `set` 一样，但是编译器不会验证存取器字段是否存在。
- `never`：不允许访问

> - **default**: Allows normal field access if the field has public visibility, otherwise equal to null access.
> - **null**: Allows access only from within the defining class.
> - **get/set**: Access is generated as a call to an accessor method. The compiler ensures that the accessor is available.
> - **dynamic**: Like get/set access, but does not verify the existence of the accessor field.
> - **never**: Allows no access at all.
>
> 定义 存取器方法： 一个类型为 `T` 的字段 `field` 的存取器方法（或简称存取器）即，一个名为 `get_field` 且类型为 `Void -> T` 的 **getter**（用于取），或是一个名为 `set_field` 且类型为 `T -> T` 的 **setter**（用于存）。
>
> [warning] Definition: Accessor method An accessor method (or short accessor) for a field named field of type T is a getter named get_field of type Void->T or a setter named set_field of type T->T.
>
> 花絮：存取器名称 在 Haxe 2中，允许任意的标识符作为访问标识符，这样就需要允许自定义的访问方法名称。这使得部分实现处理起来十分棘手。尤其是 `Reflect.getProperty()` 和 `Reflect.setProterty()` 不得不假设名称可能是任意的，因此需要目标生成器生成元信息并执行查找。
>
> 最后我们决定不再允许任意标识符，并使用 `get_` 和 `set_` 命名约定，这样做极大地简化了实现。而这也是 Haxe 2 和 Haxe 3 之间的一个断层式地改变。
>
> [warning] Trivia: Accessor names:
>
> In Haxe 2, arbitrary identifiers were allowed as access identifiers and would lead to custom accessor method names to be admitted. This made parts of the implementation quite tricky to deal with. In particular, `Reflect.getProperty()` and `Reflect.setProperty()` had to assume that any name could have been used, requiring the target generators to generate meta-information and perform lookups.
>
> We disallowed these identifiers and went for the `get_` and `set_` naming convention which greatly simplified implementation. This was one of the breaking changes between Haxe 2 and 3.

4.2.1.常见访问标识符组合

以下例子展示了属性字段常用的访问标识符组合：

> The next example shows common access identifier combinations for properties:

```
class Main {
    // 可从外部可读取，只在 Main 中写入
    public var ro(default, null):Int;

    // 可从外部写入，只在 Main 中读取
    public var wo(null, default):Int;

    // 通过 getter 和 setter 访问
    // get_x & set_x
    public var x(get, set):Int;

    // 通过 getter 读取，不可写入
    public var y(get, never):Int;

    // 字段 x 的 getter 存取器
    function get_x() return 1;

    // 字段 x 的 setter 存取器
    function set_x(x) return x;

    // 字段 y 的 getter 存取器
    function get_y() return 1;

    function new() {
        var v = x;
        x = 2;
        x += 1;
    }

    static public function main() {
        new Main();
    }
}
```

JavaScript 目标平台的输出可以帮助我们理解 main 函数中的字段访问会如何被编译：

> The JavaScript output helps understand what the field access in the main-method is compiled to:

```
var Main = function() {
    var v = this.get_x();
    this.set_x(2);
    var _g = this;
    _g.set_x(_g.get_x() + 1);
};
```

如上所述，读取访问会生成一个 `get_x()` 的调用，而写入访问会生成一个 `set_x(2)` 的调用，其中 `2` 会被赋值给 `x` 。 `+=` 操作生成的代码初看起来有些奇怪，但是可以通过下面的例子来理解：

> As specified, the read access generates a call to `get_x()` , while the write access generates a call to `set_x(2)` where `2` is the value being assigned to `x` . The way the `+=` is being generated might look a little odd at first, but can easily be justified by the following example:

```
class Main {
    public var x(get, set):Int;
    function get_x() return 1;
    function set_x(x) return x;

    public function new() { }

    static public function main() {
        new Main().x += 1;
    }
```

```
}
```

`main` 函数中对字段 `x` 的访问的表达式是复杂的：其中包含了潜在的副作用，如本例中 `Main` 的构造函数。因此，编译器不能将 `+=` 操作符直接生成为 `new Main().x = new Main().x + 1`， 而是需要把复杂的表达式保存进一个局部变量中使用：

> What happens here is that the expression part of the field access to `x` in the `main` method is **complex**: It has potential side-effects, such as the construction of `Main` in this case. Thus, the compiler cannot generate the `+=` operation as `new Main().x = new Main().x + 1` and has to cache the complex expression in a local variable:

```
Main.main = function() {
    var _g = new Main();
    _g.set_x(_g.get_x() + 1);
}
```

4.2.2.对类型系统的影响

属性字段的出现会在类型系统上产生一些影响。其中最重要的是属性字段是一个编译时特性，因此字段的类型声明在编译时必须是已知的。如果我们将一个带有属性字段的类型赋值到 `Dynamic` 对象上，那么字段访问将不再遵循存取器方法。同时，访问限制也将不再适用，此时的访问几乎都是 public 的。

> The presence of properties has several consequences on the type system. Most importantly, it is necessary to understand that properties are a compile-time feature and thus **require the types to be known**. If we were to assign a class with properties to `Dynamic`, field access would **not** respect accessor methods. Likewise, access restrictions no longer apply and all access is virtually public.

补充一个例子：

```
class Main {
    static public function main() {

         // 由于 setter 的限制，赋值小于 0 时会被钳位为 0
        trace(new Test().propertie = -100); // 输出 0

        // 因为以下访问行为定义为不可从外部访问，因此会出现编译时错误
        // trace(new Test().propertie);
         // trace(new Test().priPropertie = 200);
         // trace(new Test().priPropertie);

         // 然而赋值到 Dynamic 对象之后属性字段的访问限制不再生效
        var test:Dynamic = new Test();
        trace(test.propertie = -100); // 输出 -100
        trace(test.propertie); // 输出 -100
        trace(test.priPropertie = 200); // 输出 200
        trace(test.priPropertie); // 输出 200
    }
}

class Test {
    public function new() {}

    // 不可从外部读取，可以通过 setter 被写入
    public var propertie(null, set):Int;

    // 不可从外部被读取，也不可从外部被写入
    private var priPropertie(default, default):Int;
```

```
    function set_propertie(x) {
        if (x >= 0) {
            return x;
        } else
            return 0;
    }
}
```

编译后生成的 JavaScript：

```
// Generated by Haxe 3.4.4
(function () { "use strict";
var Main = function() { };
Main.main = function() {
    console.log(new Test().set_propertie(-100));
    var test = new Test();
    console.log(test.propertie = -100);
    console.log(test.propertie);
    console.log(test.priPropertie = 200);
    console.log(test.priPropertie);
};
var Test = function() {
};
Test.prototype = {
    set_propertie: function(x) {
        if(x >= 0) {
            return x;
        } else {
            return 0;
        }
    }
};
Main.main();
})();
```

如果属性字段使用了 `get` 或 `set` 标识符，编译器将会确保 getter 或 setter 存取器存在，否则像下面这样的代码将会产生编译错误：

> When using `get` or `set` access identifier, the compiler ensures that the getter and setter actually exists. The following code snippet does not compile:

```
class Main {
    // 缺少 x 字段所需的 get_x 方法
    public var x(get, null):Int;
    static public function main() {}
}
```

但是字段如果在子类中定义，且父类已经定义了 `get_x` 方法，那么子类中不需要再定义一次：

> The method `get_x` is missing, but it need not be declared on the class defining the property itself as long as a parent class defines it:

```
class Base {
    public function get_x() return 1;
}
class Main extends Base {
    // get_x 已经在父类中声明了，所以没问题
    public var x(get, null):Int;

    static public function main() {}
```

```
}
```

`dynamic` 访问标识符的作用与 `get` 或 `set` 一样，但是编译器不会检查存取器是否存在。

> The dynamic access modifier works exactly like get or set, but does not check for the existence

4.2.3.getter和setter的规则

存取器方法的可见性并不会影响属性本身的可见性。也就是说，如果一个属性字段被修饰为 `public` ，且定义了一个 getter ， getter 的访问标识符是否为 `private` 是无关紧要的。

> Visibility of accessor methods has no effect on the accessibility of its property. That is, if a property is `public` and defined to have a getter, that getter may be defined as `private` regardless.

getter 和 setter 都可以访问他们的物理字段进行数据存储。当字段访问表达式出现在字段本身的存取器方法中，编译器会确保此类字段访问不通过存取器方法进行，从而避免出现无限递归：

> Both getter and setter may access their physical field for data storage. The compiler ensures that this kind of field access does not go through the accessor method if made from within the accessor method itself, thus avoiding infinite recursion:

```
class Main {
   public var x(default, set):Int;

   function set_x(newX) {
       return x = newX;
   }

   static public function main() {}
}
```

然而，只有当字段的其中一个访问标识符为 `default` 或者 `null` 时，编译器才假定该物理字段存在。

> However, the compiler assumes that a physical field exists only if at least one of the access identifiers is default or null.
>
> 定义 物理字段： 一个字段若满足以下条件之一则被认为是物理字段：
>
> - 一个变量字段
> - 一个读取或写入访问为 `default` 或 `null` 的属性字段
> - 一个使用了 `:isVar` 元数据的属性字段
>
> [warning] Definition: Physical field A field is considered to be physical if it is either [warning] *a variable (4.1) [warning]* a property (4.2) with the read-access or write-access identifier being default or null [warning] * a property (4.2) with :isVar metadata (6.9)

如果一个字段不满足以上条件，那么在其存取器方法中进行字段的访问会导致编译错误：

> If this is not the case,access to the field from within an accessor method causes a compilation error:

```
class Main {
    // 字段不可以被访问，因为它不是一个实际存在的变量
    public var x(get, set):Int;

    function get_x() {
        return x;
```

```
    }

    function set_x(x) {
        return this.x = x;
    }

    static public function main() {}
}
```

如果确实需要一个物理字段，也可以通过把字段归入 `:isVar` 元数据来强制执行为物理字段：

> If a physical field is indeed intended, it can be forced by attributing the field in question with the `:isVar` metadata:

```
class Main {
    // @isVar 强制执行字段为物理字段，从而可以通过编译
    @:isVar public var x(get, set):Int;

    function get_x() {
        return x;
    }

    function set_x(x) {
        return this.x = x;
    }

    static public function main() {}
}
```

> 花絮 属性的 setter 类型：
>
> 第一次使用 Haxe 的人通常会疑惑为什么 setter 的类型是 `T -> T` 而不是 `T -> Void` ，也就是为什么 setter 需要返回一个值？
>
> 原因是我们想要在给字段赋值时，setter 可以作为 右手表达式（right-side expression) 使用。从而使用诸如 `x = y = 1` 这样的链式的语法结构，这种链式结构会被解析为 `x = (y = 1)` 。而为了能够把 `y = 1` 的结果赋值给 `x` ，表达式必须能够返回一个值，而如果说 `y` 有一个 setter 并且会返回 `Void` ，那么这种链式的结构就没有意义了。
>
> [warning] Trivia: Property setter type It is not uncommon for new Haxe users to be surprised by the type of a setter being required to be T->T instead of the seemingly more natural T->Void. After all, why would a setter have to return something? The rationale is that we still want to be able to use field assignments using setters as rightside expressions. Given a chain like x = y = 1, it is evaluated as x = (y = 1). In order to assign the result of y = 1 to x, the former must have a value. If y had a setter returning Void, this would not be possible.

4.3.方法

变量保存数据，方法通过托管表达式（第5章）来定义程序的行为。我们已经在每个代码例子中看过方法字段，前面的 HelloWorld示例就包含一个 main 方法：

> While variables (4.1) hold data, methods are defining behavior of a program by hosting expressions (5). We have seen method fields in every code example of this document with even the initial Hello World (1.3) example containing a main method:

```
class Main {
    static public function main():Void {
        trace("Hello World");
    }
}
```

方法通过 function 关键字识别。我们还可以了解到，它们：

> Methods are identified by the function keyword. We can also learn that they

1. 有一个名字（这里是main）
2. 有一个参数列表（这里为 empty()）
3. 有一个返回类型（这里是 Void）
4. 可能有访问修饰符（第4.4节）（这里是 static 和public）
5. 可能有一个表达式（这里是 {trace("Hello World");}）

> 1. have a name (here: main),
> 2. have an argument list (here: empty ()),
> 3. have a return type (here: Void),
> 4. may have access modifiers (4.4) (here: static and public) and
> 5. may have an expression (here: {trace("Hello World");}).

还可以看下面的例子，了解更多参数和返回类型的知识：

> We can also look at the next example to learn more about arguments and return types:

```
class Main {
    static public function main() {
        myFunc("foo", 1);
    }

    static function myFunc(f:String, i) {
        return true;
    }
}
```

参数通过字段名后一个开口的 (括号开始，一个 逗号 , 作为参数列表中每个参数的分隔符号，然后跟一个闭口的) 括号。参数规范的附加信息在 函数类型（第2.6节）中描述。

> Arguments are given by an opening parenthesis (after the field name, a comma , separated list of argument specifications and a closing parenthesis). Additional information on the argument specification is described in Function Type (Section 2.6).

例子展示了类型推断如何被使用到两个参数和返回类型上。方法 myFunc 有两个参数，但是第一个被显式赋予类型，f，为String类型。第二个参数 i ，没有类型示意，留给编译器从它的调用中推断它的类型。此外，返回类型通过return ture 表达式来推断为 Bool 。

> The example demonstrates how type inference(3.6) can be used for both argument and return types. The method myFunc has two arguments but only explicitly gives the type of the first one, f, as String. The second one, i, is not type-hinted and it is left to the compiler to infer its type from calls made to it. Likewise,the return type of the method is inferred from the return true expression as Bool.

4.3.1.重写方法

重写字段是创建类的层级的结构。许多设计模式使用到它，但是这里我们只探索基本的功能。为了在类中使用重写，需要这个类有一个父类（第2.3.2）。思考下面的例子：

> Overriding fields is instrumental for creating class hierarchies. Many design patterns utilize it, but here we will explore only the basic functionality. In order to use overrides in a class, it is required that this class has a parent class (2.3.2). Let us consider the following example:

```
 class Base {
    public function new() { }
    public function myMethod() {
        return "Base";
    }
}

class Child extends Base {
    public override function myMethod() {
        return "Child";
    }
}

class Main {
    static public function main() {
        var child:Base = new Child();
        trace(child.myMethod()); // Child
    }
}
```

这里重要的组件是：

> The important components here are:

- Base 类，有一个方法 myMethod和一个构造函数
- Child类，继承Base类也有一个方法 myMethod，通过 override关键字声明
- Main 类，它的main方法创建一个Child类的实例，分配它到一个变量 child ，显式的声明类型为Base，然后在其上调用 myMethod()

> - the class Base which has a method myMethod and a constructor,
> - the class Child which extends Base and also has a method myMethod being declared with override, and
> - the Main class whose main method creates an instance of Child, assigns it to a variable child of explicit type Base and calls myMethod() on it.

变量 child 被显式的类型化为 Base 来突出一个重要的不同：在编译器时类型被认为是 Base，但是运行时仍然查找正确的方法即类Child中的 myMethod。这是因为字段访问是在运行时动态解析的。

> The variable child is explicitly typed as Base to highlight an important difference: At compile-time the type is known to be Base,but the runtime still finds the correct method myMethod on class Child. This is because field access is resolved dynamically at runtime.

Child 类可以访问它重载的方法，通过调用 super.methodName()：

> The Child class can access methods it has overriden by calling super.methodName():

```
 class Base {
    public function new() { }
    public function myMethod() {
        return "Base";
    }
}

class Child extends Base {
    public override function myMethod() {
        return "Child";
    }

    public function callHome() {
        return super.myMethod();
    }
}

class Main {
    static public function main() {
        var child = new Child();
        trace(child.callHome()); // Base
    }
}
```

继承（第2.3.2节）中解释了super()在一个新的构造函数中的使用。

> The section on Inheritance (Section 2.3.2) explains the use of super() from within a new constructor.

4.3.2.变异和访问修饰符的影响

重载遵守变异（第3.4节）的规则。也就是说，它们的参数类型允许逆变（不那么特定的类型）而它们的返回类型允许共变（更特别的类型）。

> Overriding adheres to the rules of variance (3.4). That is, their argument types allow contravariance (less specific types) while their return type allows covariance (more specific types):

```
class Base {
    public function new() { }
}

class Child extends Base {
    private function method(obj:Child):Child {
        return obj;
    }
}

class ChildChild extends Child {
    public override function method(obj:Base):ChildChild {
        return null;
    }
}
```

```
class Main {
    static public function main() { }
}
```

直观的说，这是因为参数被“写入”到函数中，而返回值是从函数中“读取”。

> Intuitively, this follows from the fact that arguments are “written to” the function and the return value is “read from” it.

示例也展示了如何改变可见性（第4.4.1节）：一个重载的字段可能是 public ，如果被重载的字段是private，但是相反则不行。

> The example also demonstrates how visibility (4.4.1) may be changed: An overriding field may be public if the overridden field is private, but not the other way around.

不可能重载声明为内联（第4.4.2节）的字段。这是由于冲突的概念：当内联在编译时通过替换一个函数体的调用，重载字段必须被在运行时决定。

> It is not possible to override fields which are declared as inline (4.4.2). This is due to the conflicting concepts: While inlining is done atcompile-time by replacinga call with thefunction body, overriding fields necessarily have to be resolved at runtime.

4.4.访问修饰符

本节内容：

- 可见性
- inline修饰符
- dynamic修饰符
- override修饰符
- static修饰符

4.4.1.可见性

字段默认为 private ，意味着只有类和它的子类可以访问它们。它们可以被声明为公共字段，通过使用 public 访问修饰符，使它们可以在各处访问。

> Fields are by default private, meaning that only the class and its sub-classes may access them. They can be made public by using the public access modifier, allowing access from anywhere.

```
 class MyClass {
    static public function available() {
        unavailable();
    }
    static private function unavailable() { }
}

class Main {
    static public function main() {
        MyClass.available();
        // Cannot access private field unavailable
        MyClass.unavailable();
    }
}
```

MyClass类的可用字段的访问允许从Main中访问，因为它表示为 public。然而，当访问不可用的字段访问可以从类MyClass内部，但是不能从Main中访问，因为它是private（明确的私有声明，尽管这个标识符在这里是多余的）

> Access to field available of class MyClass is allowed from within Main because it is denoted as being public. However, while access to field unavailable is allowed from within class MyClass, it is not allowed from within class Main because it is private (explicitly, although this identifier is redundant here).

例子展示了static字段的可见性，但是成员字段的规则是等价的。下面的示例展示了当继承（第2.3.2节）被使用时的可见性行为。

> The example demonstrates visibility through static fields, but the rules for member fields are equivalent. The following example demonstrates visibility behavior for when inheritance (2.3.2) is involved.

```
class Base {
    public function new() { }
    private function baseField() { }
}

class Child1 extends Base {
    private function child1Field() { }
```

```
}

class Child2 extends Base {
    public function child2Field() {
        var child1 = new Child1();
        child1.baseField();
        // Cannot access private field child1Field
        child1.child1Field();
    }
}

class Main {
    static public function main() { }
}
```

我们可以看到访问 child1.baseField() 是允许在Child2类中访问，即使child1是不同的类型，Child1 。这是因为字段被定义在它们的通用祖先类Base，相反的，字段 Child1Field不能被从Child2中访问。

> We can see that access to child1.baseField() is allowed from within Child2 even though child1 is of a different type,Child1. This is because the field is defined on their common ancestor class Base, contrary to field child1Field which can not be accessed from within Child2.

省略可见性的修饰符通常默认可见性为 private，但是有例外它会变成 public：

> Omitting the visibility modifier usually defaults the visibility to private, but there are exceptions where it becomes public instead:

1. 如果类被声明为 extern
2. 如果字段被声明在一个接口（第2.3.3节）
3. 如果字段重载（第4.3.1节）了一个 public 字段

> 1. If the class is declared as extern.
> 2. If the field is declared on an interface (2.3.3).
> 3. If the field overrides (4.3.1) a public field.

Protected

> 花絮：Protected Haxe没有类似Java和C++等其它面向对象语言中的protected概念。然而，它的private行为等同于那些语言的protected行为，所以Haxe实际上缺少的是那些语言中的 private 行为。 [warning] **Trivia**: Protected Haxe has no notion of a protected keyword known from Java, C++ and other object-oriented languages. However, its private behavior is equal to those language's protected behavior, so Haxe actually lacks their real private behavior.

4.4.2.Inline

inline 关键字允许函数体被直接插入到它们调用位置。这是一个强大的优化工具，但是应该审慎使用，并不是所有函数都适用inline行为。下面的例子演示了基本的用法：

> The inline keyword allows function bodies to be directly inserted in place of calls to them. This can be a powerful optimization tool,but should be used judiciously as not all functions are good candidates for inline behavior. The following example demonstrates the basic usage:

```
class Main {
    static inline function mid(s1:Int, s2:Int) {
        return (s1 + s2) / 2;
    }
```

```
    static public function main() {
        var a = 1;
        var b = 2;
        var c = mid(a, b);
    }
}
```

生成的JavaScript输出揭示了内联的效果：

> The generated JavaScript output reveals the effect of inline:

```
(function () { "use strict";
    var Main = function() { }
    Main.main = function() {
        var a = 1;
        var b = 2;
        var c = (a + b) / 2;
    }
    Main.main();
})();
```

显然，字段mid被生成的函数体(s1 + s2)/2替换掉了调用 mid(a,b)，s1被替换为 a ，s2被替换为b。这可以避免一个函数调用，根据目标和出现的频率，可以产生显著的性能改进。

> As evident, the function body (s1 + s2) / 2 of field mid was generated in place of the call to mid(a, b), with s1 being replaced by a and s2 being replaced by b. This avoids a function call which,depending on the target and frequency of occurrences,may yield noticeable performance improvements.

并不总是容易判断是否一个函数要限定为内联函数。没有编写表达式的短函数（如a=形式的赋值）通常是一个好的选择，但是有时候更复杂的函数也可以使用内联。然而，在一些情况下内联可以实际上损害部分性能，例如，因为编译器必须创建临时变量服务于复杂的表达式。

> It is not always easy to judge if a function qualifies for being inline. Short functions that have no writing expressions (such as a = assignment) are usually a good choice, but even more complex functions can be candidates. However, in some cases inlining can actually be detrimental to performance, e.g. because the compiler has to create temporary variables for complex expressions.

内联并不保证执行。编译器可能由于多种原因取消内联，用户也可以通过 --no-inline 命令行参数来禁止内联。唯一的例外是如果类是 extern（第6.2节）或者如果泪字段有 :extern 元数据（第6.9节），这种情况内联被禁止。如果她不能被执行，编译器发出一个错误。

> Inline is not guaranteed to be done. The compiler might cancel inlining for various reasons or a user could supply the --no-inline command line argument to disable inlining. The only exception is if the class is extern (6.2) or if the class field has the :extern metadata (6.9), in which case inline is forced. If it cannot be done, the compiler emits an error.

重要的是依赖内联时要记得这个：

> It is important to remember this when relying on inline:

```
class Main {
    public static function main () { }

    static function test() {
        if (Math.random() > 0.5) {
            return "ok";
        } else {
```

```
            error("random failed");
        }
    }

    @:extern static inline function error(s:String) {
        throw s;
    }
}
```

如果正确的调用error是内联的，程序编译正确，因为控制流检查器满意内联的throw（第5.22节）表达式。如果内联部执行，编译器只发现一个error函数的调用并发出错误 A return is missing here。

> If the call to error is inlined the program compiles correctly because the control flow checker is satisfied due to the inlined throw (5.22) expression. If inline is not done,the compiler only sees a function call to error and emits the error A return is missing here.

Inline variables

The `inline` keyword can also be applied to variables, but only when used together with `static`. An inline variable must be initialized to a constant, otherwise the compiler emits an error. The value of the variable is used everywhere in place of the variable itself.

The following code demonstrates the usage of an inline variable:

```
class Main {
  static inline final language = "Haxe";

  static public function main() {
    trace(language);
  }
}
```

The generated JavaScript shows that the `language` variable is not present anymore:

```
(function ($global) { "use strict";
var Main = function() { };
Main.main = function() {
    console.log("root/program/Main.hx:5:","Haxe");
};
Main.main();
})({});
```

Note that even though we call such kind of fields "variables", inline variables can never be reassigned as the value must be known at compile-time to be inlined at the place of usage. This makes inline variables a subset of `final` fields, hence the usage of the `final` keyword in the code example above.

> **Trivia:** `inline var`
>
> Prior to Haxe 4, there was no `final` keyword. The inline variables feature however was present for a long time, using the `var` keyword instead of `final`. Using `inline var` still works in Haxe 4 but might be deprecated in the future, because `final` is more appropriate.

4.4.3.Dynamic

方法可以通过 dynamic 关键字 使它们可绑定（重绑定）：

> Methods can be denoted with the dynamic keyword to make them (re-)bindable:

```
class Main {
    static dynamic function test() {
        return "original";
    }
    static public function main() {
        trace(test()); // original
        test = function() { return "new"; }
        trace(test()); // new
    }
}
```

第一次调用 test() 调用了原来的函数，返回字符串 “original”。下一行，test被分配了一个新的函数。这恰恰是dynamic做到的：函数字段可以被分配一个新的函数。其结果是，下一次调用 test() 返回字符串 “new” 。

> The first call to test() invokes the original function which returns the String "original". In the next line, test is assigned a new function. This is precisely what dynamic allows: Function fields can be assigned a new function. As a result,the next invocation of test() returnsthe String "new".

动态字段因为明显的理由不能内联：内联是在编译时执行，动态函数必须被在运行时决定。

> Dynamic fields cannot be inline for obvious reasons: While inlining is done at compiletime, dynamic functions necessarily have to be resolved at runtime.

4.4.4.Override

访问修饰符 override 要求被修饰的字段声明也同时出现在父类（第2.3.2节）中。它的目的是确保类的作者意识到 override 因为这不能总是被明显的在庞大的类层级中。此外，如果字段修饰了override关键字而实际上没有重写任何东西（例如由于字段名的拼写错误导致）会引发一个错误。

> The access modifier override is required when a field is declared which also exists on a parent class(2.3.2). Its purpose is to ensure that the author of a class is aware of the override as this may not always be obvious in large class hierarchies. Likewise, having override on a field which does not actually override anything (e.g. due to a misspelled field name) triggers an error.

重载字段的效果在重载方法（第4.3.1节）详述。这个修饰符只允许用于方法（第4.3节）字段。

> The effects of overriding fields are detailed in Overriding Methods (Section 4.3.1). This modifier is only allowed on method (4.3) fields.

4.4.5.Static

所有的字段除非修饰符标记为 static 否则都是成员字段。Static字段用在类上，就像非静态字段用在类的实例上。

所有的字段除非修饰符标记为 static 否则都是成员字段。static字段通过类使用，而非static字段通过类的实例使用。

> All fields are member fields unless the modifier static is used. Static fields are used “on the class” whereas non-static fields are used “on a class instance”:

```
class Main {
    static function main() {
        Main.staticField; // static read
```

```
        Main.staticField = 2; // static write
    }
    static var staticField:Int;
}
```

静态变量（第4.1节）和属性（第4.2节）字段可以使用任意的初始化表达式（第5章）.

> Static variable (4.1) and property (4.2) fields can have arbitrary initialization expressions (5).

5.表达式

Haxe中的表达式定义了程序需要做什么。多数表达式可以在方法（第4.3节）体中找到，它们被结合来描述方法的功能。本节解释不同类型的表达式。一些定义可以提供帮助：

> Expressions in Haxe define what a program does. Most expressions are found in the body of a method (4.3), where they are combined to express what that method should do. This section explains the different kinds of expressions. Some definitions help here:

Name

> 定义：Name 一个名称通常指代
>
> - 一个类型
> - 一个局部变量
> - 一个局部函数，或者
> - 一个字段
>
> **Definition**: Name A general name may refer to
>
> - a type,
> - a local variable,
> - a local function or
> - a field.

Identifier

> 定义：标识符 Haxe的标识符由一个下划线、一个美元符号*$*，一个小写字母*a-z*或者一个大写字母*A-Z*开始。后可接任意数量的数字、下划线、大小写字母的组合。更多的限制受制于使用的上下文，根据类别进行检查： 类型名必须使用大写的字母或者一个下划线开始。 美元符号不允许用于任何类别的名字的开头（美元名字多数用于宏的具体化（第*9.3*节））。 *[warning]* ***Definition****: Identifier Haxe identifiers start with an underscore* , a dollar $, a lower-case character a-z or an upper-case character A-Z. After that, any combination and number of _, A-Z, a-z and 0-9 may follow. Further limitations follow from the usage context, which are checked upon typing:
>
> - Type names must start with an upper-case letter A-Z or an underscore _.
> - Leading dollars are not allowed for any kind of name(5)(dollar-names are mostly used for macro reification (9.3)).

Since Haxe 3.3.0

Haxe保留前置 *hx* 的标识符在内部使用。这并不是强制的被解析器或者类型检查器执行。

> Haxe reserves the identifier prefix *hx* for internal use. This is not enforced by the parser or typer.

关键字：下面的Haxe关键字不能被使用为标识符：

> **Keywords** The following Haxe keywords may not be used as identifiers:

- abstract
- break
- case
- cast
- catch

- class
- continue
- default
- do
- dynamic
- else
- enum
- extends
- extern
- false
- for
- function
- if
- implements
- import
- in
- inline
- interface
- macro
- new
- null
- override
- package
- private
- public
- return
- static
- switch
- this
- throw
- true
- try
- typedef
- untyped
- using
- var
- while

5.1.块

Haxe中的一个块由一个开口的 花括号 { 开始，以一个闭口的花括号 }结束。一个块可以包含一些表达式，每个使用分号结束。通常的语法是：

> A block in Haxe starts with an opening curly brace { and ends with a closing curly brace }. A block may contain several expressions, each of which is followed by a semicolon ;. The general syntax is thus:

```
{
    expr1;
    expr2;
    ...
    exprN;
}
```

被块表达式扩展的值和类型等于最后一个子表达式的值和类型。

> The value and by extension the type of a block-expression is equal to the value and the type of the last sub-expression.

块可以包含局部变量，通过 var表达式（第5.10节）声明，同样，局部函数通过 function 表达式（第5.11节）声明。它们在块和子块中中是可用的，但是不能在块的范围之外使用。同样，只有在被声明之后才可以使用。下面的例子使用 var，但是同样的规则也适用于 function 的使用：

> Blocks can contain local variables declared by var expression (5.10), as well as local functions declared by function expressions (5.11). These are available within the block and within sub-blocks, but not outside the block. Also, they are available only after their declaration. The following example uses var, but the same rules apply to function usage:

```
{
    a; // error, a is not declared yet
    var a = 1; // declare a
    a; // ok, a was declared
    {
        a; // ok, a is available in sub-blocks
    }
    // ok, a is still available after
    // sub-blocks
    a;
}
a; // error, a is not available outside
```

在运行时，块从头至尾执行。控制流（如异常（第5.18节）或者返回表达式（第5.19节））可能在所有的表达式被执行之前离开块。

> At runtime, blocks are evaluated from top to bottom. Control flow (e.g. exceptions (5.18) or return expressions (5.19)) may leave a block before all expressions are evaluated.

5.2.常量

Haxe语法支持下面的常量：

> The Haxe syntax supports the following constants:

Int：一个整型（第2.1.1节），例如0，1，97121，-12， 0xFF0000 。 **Float**：一个浮点数（第2.1.1节），如0.0，1.，.3，-93.2 。 **String**：一个字符串（第10.1节），如""，“foo”，''，'bar' 。 **true**，**false**：一个布尔值 **null**：null值

> **Int**: An integer (2.1.1), such as 0, 1, 97121, -12, 0xFF0000. **Float**: A floating point number (2.1.1), such as 0.0, 1., .3, -93.2. **String**: A string of characters (10.1), such as "", "foo", '', 'bar'. **true,false**: A boolean (2.1.4) value. **null**: The null value.

此外，内部的语法结构将标识符作为常量，可能在使用宏（第9章）时是相关的。

> Furthermore, the internal syntax structure treats identifiers (5) as constants, which may be relevant when working with macros (9).

Constants are values which are immutable. These values can be used as inline variables and default values for function arguments. All constants are literals, except for argument-less enum constructors:

Example	Type	Note
`42` , `0xFF42`	`Int`	integer constant
`0.32` , `3.` , `2.1e5`	`Float`	floating-point decimal constant
`true` , `false`	`Bool`	boolean constant
`~/haxe/gi`	`EReg`	regular expression
`null`	`T`	null value for any nullable type
`"XXX"` , `'XXX'`	`String`	string literal
`"X".code` , `'X'.code`	`Int`	Unicode character codepoint
`MyEnum.Haxe`	`T`	enum constructor with no arguments

Furthermore, the internal syntax structure treats identifiers as constants, which may be relevant when working with macros.

5.3 操作符

5.3.1 一元操作符

Operator	Operation	Operand type	Position	Result type
`~`	bitwise negation	`Int`	prefix	`Int`
`!`	logical negation	`Bool`	prefix	`Bool`
`-`	arithmetic negation	`Float/Int`	prefix	same as operand
`++`	increment	`Float/Int`	prefix and postfix	same as operand
`--`	decrement	`Float/Int`	prefix and postfix	same as operand

Increment and decrement

递增与递减操作符用于改变一个给定的值，因此不能被用于一个 “只读” 的值。同时他们可以根据前置或后置方式的使用产生不同的结果，前置使用时求值（evaluates）结果为修改后的值，后置使用时求值结果为原来的值。

The increment and decrement operators change the given value, so they cannot be applied to a read-only value. They also produce different results based on whether they are used as a prefix operator, which evaluates to the modified value, or as a postfix operator, which evaluates to the original value:

```
var a = 10;
trace(a++); // 10
trace(a); // 11

a = 10;
trace(++a); // 11
trace(a); // 11
```

5.3.2 二元操作符

Arithmetic operators

Operator	Operation	Operand 1	Operand 2	Result type
`%`	modulo	`Float/Int`	`Float/Int`	`Float/Int`
`*`	multiplication	`Float/Int`	`Float/Int`	`Float/Int`
`/`	division	`Float/Int`	`Float/Int`	`Float`

`+`	addition	`Float/Int`	`Float/Int`	`Float/Int`
`-`	subtraction	`Float/Int`	`Float/Int`	`Float/Int`

About the `Float/Int` return type: If one of the operands is of type `Float`, the resulting expression will also be of type `Float`, otherwise the type will be `Int`. The result of a division is always a `Float`; use `Std.int(a / b)` for integer division (discarding any fractional part).

In Haxe, the result of a modulo operation always keeps the sign of the dividend (the left operand) if the divisor is non-negative. The result is target-specific with a negative divisor.

String concatenation operator

Operator	Operation	Operand 1	Operand 2	Result type
`+`	concatenation	any	`String`	`String`
`+`	concatenation	`String`	any	`String`
`+=`	concatenation	`String`	any	`String`

Note that the "any" operand will be stringified. For classes and abstracts stringification can be controlled with user-defined `toString` function.

Bitwise operators

Operator	Operation	Operand 1	Operand 2	Result type
`<<`	shift left	`Int`	`Int`	`Int`
`>>`	shift right	`Int`	`Int`	`Int`
`>>>`	unsigned shift right	`Int`	`Int`	`Int`
`&`	bitwise and	`Int`	`Int`	`Int`
`|`	bitwise or	`Int`	`Int`	`Int`
`^`	bitwise xor	`Int`	`Int`	`Int`

Logical operators

Operator	Operation	Operand 1	Operand 2	Result type
`&&`	logical and	`Bool`	`Bool`	`Bool`
`||`	logical or	`Bool`	`Bool`	`Bool`

Short-circuiting:

Haxe guarantees that compound boolean expressions with the same operator are evaluated from left to right but only as far as necessary at run-time. For instance, an expression like `A && B` will evaluate `A` first and evaluate `B` only if the evaluation of `A` yielded `true`. Likewise, the expression `A || B` will not evaluate `B` if the evaluation of `A` yielded `true`, because the value of `B` is irrelevant in that case. This is important in cases such as this:

```
if (object != null && object.field == 1) { }
```

Accessing `object.field` if `object` is `null` would lead to a run-time error, but the check for `object != null` guards against it.

Compound assignment operators

Operator	Operation	Operand 1	Operand 2	Result type
`%=`	modulo	`Float/Int`	`Float/Int`	`Float/Int`
`*=`	multiplication	`Float/Int`	`Float/Int`	`Float/Int`
`/=`	division	`Float`	`Float/Int`	`Float`
`+=`	addition	`Float/Int`	`Float/Int`	`Float/Int`
`-=`	subtraction	`Float/Int`	`Float/Int`	`Float/Int`
`<<=`	shift left	`Int`	`Int`	`Int`
`>>=`	shift right	`Int`	`Int`	`Int`
`>>>=`	unsigned shift right	`Int`	`Int`	`Int`
`&=`	bitwise and	`Int`	`Int`	`Int`
`|=`	bitwise or	`Int`	`Int`	`Int`
`^=`	bitwise xor	`Int`	`Int`	`Int`

In all cases, a compound assignment modifies the given variable, field, structure member, etc., so it will not work on a read-only value. The compound assignment evaluates to the modified value when used as a sub-expression:

```
var a = 3;
trace(a += 3); // 6
trace(a); // 6
```

Note that the first operand of `/=` must always be a `Float`, since the result of a division is always a `Float` in Haxe. Similarly, `+=` and `-=` cannot accept `Int` as the first operand if `Float` is given as the second operand, since the result would be a `Float`.

Numeric comparison operators

Operator	Operation	Operand 1	Operand 2	Result type
`==`	equal	`Float/Int`	`Float/Int`	`Bool`
`!=`	not equal	`Float/Int`	`Float/Int`	`Bool`
`<`	less than	`Float/Int`	`Float/Int`	`Bool`
`<=`	less than or equal	`Float/Int`	`Float/Int`	`Bool`
`>`	greater than	`Float/Int`	`Float/Int`	`Bool`
`>=`	greater than or equal	`Float/Int`	`Float/Int`	`Bool`

String comparison operators

Operator	Operation	Operand 1	Operand 2	Result type

`==`	equal	`String`	`String`	`Bool`
`!=`	not equal	`String`	`String`	`Bool`
`<`	lexicographically before	`String`	`String`	`Bool`
`<=`	lexicographically before or equal	`String`	`String`	`Bool`
`>`	lexicographically after	`String`	`String`	`Bool`
`>=`	lexicographically after or equal	`String`	`String`	`Bool`

Two values of type `String` are considered equal in Haxe when they have the same length and the same contents:

```
var a = "foo";
var b = "bar";
var c = "foo";
trace(a == b); // false
trace(a == c); // true
trace(a == "foo"); // true
```

Equality operators

Operator	Operation	Operand 1	Operand 2	Result type
`==`	equal	any	any	`Bool`
`!=`	not equal	any	any	`Bool`

The types of operand 1 and operand 2 must unify.

Enums:

- Enums without parameters always represent the same value, so `MyEnum.A == MyEnum.A`.
- Enums with parameters can be compared with `a.equals(b)` (which is short for `Type.enumEquals()`).

Dynamic:

Comparison involving at least one operand of type `Dynamic` is unspecified and platform-specific.

Miscellaneous operators

Operator	Operation	Operand 1	Operand 2	Result type
`...`	interval (see range iteration)	`Int`	`Int`	`IntIterator`
`=>`	arrow (see map, key-value iteration, map comprehension)	any	any	-

5.3.3 三元操作符

Operator	Operation	Operand 1	Operand 2	Operand 3	Result type
`?:`	condition	`Bool`	any	any	any

The type of operand 1 and operand 2 must unify. The unified type is used as the result type of the expression.

The ternary conditional operator is a shorter form of `if`:

```
trace(true ? "Haxe" : "Neko"); // Haxe
trace(1 == 2 ? 3 : 4); // 4

// equivalent to:

trace(if (true) "Haxe" else "Neko"); // Haxe
trace(if (1 == 2) 3 else 4); // 4
```

5.3.4 优先级

In order of descending precedence (i.e. operators higher in the table are evaluated first):

Operators	Note	Associativity
`!`, `++`, `--`	postfix unary operators	right
`~`, `!`, `-`, `++`, `--`	prefix unary operators	right
`%`	modulo	left
`*`, `/`	multiplication, division	left
`+`, `-`	addition, subtraction	left
`<<`, `>>`, `>>>`	bitwise shifts	left
`&`, `|`, `^`	bitwise operators	left
`==`, `!=`, `<`, `<=`, `>`, `>=`	comparison	left
`...`	interval	left
`&&`	logical and	left
`||`	logical or	left
`@`	metadata	right
`?:`	ternary	right
`%=`, `*=`, `/=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `|=`, `^=`	compound assignment	right
`=>`	arrow	right

Differences from C-like precedence

Many languages (C++, Java, PHP, JavaScript, etc) use the same operator precedence rules as C. In Haxe, there are a couple of differences from these rules:

- `%` (modulo) has a higher precedence than `*` and `/`; in C they have the same precedence
- `|`, `&`, `^` (bitwise operators) have the same precedence; in C the three operators all have a different precedence

- `|`, `&`, `^` (bitwise operators) also have a lower precedence than `==`, `!=`, etc (comparison operators)

5.3.5 重载与宏

前面章节中所描述的操作符规定了其作用于基础数据类型时的意义及返回类型。 额外的功能可通过 抽象操作符重载 或 宏处理 实现。

The operators specified in the previous sections specify the types and meanings for operations on basic types. Additional functionality can be implemented using abstract operator overloading or macro processing.

操作符优先级不可以通过抽象操作符重载改变。

Operator precedence cannot be changed with abstract operator overloading.

宏处理的某些特定时候有一个额外的操作符可供使用：后置的 `!` 操作符。

For macro processing in particular, there is an additional operator available: the postfix `!` operator.

5.5.数组声明

数组使用封闭的方括号，并在其中使用逗号分隔值来初始化。一个扁平的 [] 表示空数组，而[1,2,3]用三个元素初始化一个数组。

> Arrays are initialized by enclosing comma , separated values in brackets []. A plain [] represents the empty array, whereas [1, 2, 3] initializes an array with three elements 1, 2 and 3.

生成的代码可能在不支持数组初始化的平台不那么简洁。本质上，这样的初始化代码看起来如下：

> The generated code may be less concise on platforms that do not support array initialization. Essentially, such initialization code then looks like this:

```
var a = new Array();
a.push(1);
a.push(2);
a.push(3);
```

在声明时要考虑这个，是否一个函数应该被内联（第4.4.2节），因为它可能内联进比在可见语法中更多的代码。

> This should be considered when deciding if a function should be inlined (4.4.2) as it may inline more code than visible in the syntax.

高级初始化技术在数组推导（第6.6节）中详述。

> Advanced initialization techniques are described in Array Comprehension (Section 6.6).

5.6.对象声明

对象声明通过一个开口的花括号，后跟键值对，通过逗号分隔，然后，使用闭口的花括号结束。

> Object declaration begins with an opening curly brace { after which key:value-pairs separated by comma , follow, and which ends in a closing curly brace}.

```
{
    key1:value1,
    key2:value2,
    ...
    keyN:valueN
}
```

对象声明更多的细节在匿名结构（第2.5节）中详述。

> Further details of object declaration are described in the section about anonymous structures (2.5).

5.7.字段访问

字段访问通过点操作符来表示，点号后跟字段的名称。

> Field access is expressed by using the dot . followed by the name of the field.

```
object.fieldName
```

这个语法同样用于访问包中的类型，形式为 pack.Type 。

> This syntax is also used to access types within packages in the form of pack.Type.

类型检查器确保一个被访问的字段实际上存在，并可以应用根据字段的性质进行转换。如果一个字段访问是模糊的，理解解析顺序（第3.7.3）可以提供帮助。

> The typer ensures that an accessed field actually exist and may apply transformations depending on the nature of the field. If a field access is ambiguous, understanding the resolution order (3.7.3) may help.

5.8.数组访问

数组访问通过使用开口的 [方括号，后跟索引表达式 ，并以闭口的方括号]结束。

> Array access is expressed by using an opening bracket [followed by the index expression and a closing bracket].

```
expr[indexExpr]
```

这个符号允许使用任意表达式，但是在类型级别，只有某个组合被承认：

> This notation is allowed with arbitrary expressions, but at typing level only certain combinations are admitted:

- expr 是Array或者 Dynamic 类型，indexExpr是Int类型
- expr是一个抽象类型（第2.8节），定义要给匹配的数组访问（第2.8.3节）

> - expr is of Array or Dynamic and indexExpr is of Int
> - expr is an abstract type (2.8) which defines a matching array access (2.8.3)

5.9.函数调用

函数调用由一个任意的主题表达式，后跟要给开口的圆括号，逗号分隔表达式作为参数，以闭口的圆括号结束。

> Functions calls consist of an arbitrary subject expression followed by an opening parenthesis (, a comma , separated list of expressions as arguments and a closing parenthesis).

```
subject(); // call with no arguments
subject(e1); // call with one argument
subject(e1, e2); // call with two arguments
// call with multiple arguments
subject(e1, ..., eN);
```

5.10.var

var 关键字允许声明多个变量，通过逗号分隔。每个变量有一个有效的标识符（第5章），可选的可以有一个值分配，跟在赋值操作符 = 之后。变量也可以有一个显式的类型标记。

> The var keyword allows declaring multiple variables, separated by comma ,. Each variable has a valid identifier (5) and optionally a value assignment following the assignment operator =. Variables can also have an explicit type-hint.

```
var a; // declare local a
var b:Int; // declare variable b of type Int
// declare variable c, initialized to value 1
var c = 1;
// declare variable d and variable e
// initialized to value 2
var d,e = 2;
```

局部变量的作用于行为在块（第5.1节）中描述。

> The scoping behavior of local variables is described in Blocks (Section 5.1).

5.11.局部函数

Haxe 支持表达式中使用 类级 函数声明也支持 局部 函数声明。其语法遵循类字段方法（4.3）的语法：

Haxe supports first-class functions and allows declaring local functions in expressions. The syntax follows class field methods (4.3):

```
class Main {
    static public function main() {
        var value = 1;
        function myLocalFunction(i) {
            return value + i;
        }
        trace(myLocalFunction(2)); // 3
    }
}
```

我们声明了一个 `myLocalFunction` 位于类字段 `main` 的块表达式（5.1）中。它接受一个参数，并对参数加上外部作用域中定义的 `value` 的值。

We declare myLocalFunction inside the block expression (5.1) of the main class field. It takes one argument i and adds it to value, which is defined in the outside scope.

其作用域等同于 变量（5.10）的作用域，且大多数情况下声明一个带命名的局部函数实际上可以被看作是把一个匿名函数赋值到一个变量上：

The scoping is equivalent to that of variables (5.10) and for the most part writing a named local function can be considered equal to assigning an unnamed local function to a local variable:

```
var myLocalFunction = function(a) { }
```

但是当涉及类型参数（3.2）以及函数所处位置时有些不同。当一个函数在声明时没有被赋值给任何变量，我们称它为 'lvalue' 函数，反之称其为 'rvalue' 函数

- lvalue 函数需要一个名字，且可以具有类型参数（3.2）
- rvalue 函数不一定有名字，但不可以具有类型参数

However, there are some differences related to type parameters and the position of the function. We speak of a “lvalue” function if it is not assigned to anything upon its declaration, and an “rvalue” function otherwise.

- Lvalue functions require a name and can have type parameters (3.2).
- Rvalue functions may have a name, but cannot have type parameters.

```
var func = function(){}; // 这是一个 rvalue 函数，它有名字，可以通过 func 这个“名字”来调用它
function(){}; // 同样这也是一个 rvalue 函数，但是它没有名字，是个孤儿，但是除非作为某个函数参数，否则单独这样放在某个语句块会报错。
function hasName <T> (){}; // 这是一个 lvalue 函数可以具有类型参数

var funFact = function gotNamed<T>(x:T){ trace('T is: ${x}'); }; // 这样的声明不可行，尽管 gotNamed 是 lvalue 但是编译器会认为 funFact 实际上是作为 rvalue 使用，所以编译器会抛出错误提示；所以其实这种写法并不能作为某个参数来传递
var funFact = hasName; // 但是这样可以， 此时相当于 lvalue 函数 hasName 的另外一个别名为 funFact 。。。
```

5.12.new

new 关键字表示一个类（第2.3节）或者一个抽象类型（第2.8节）被实例化。后面跟着将要被实例化的类型的类型路径（第3.7节）。它也可以在 `<>` 中显式地列出类型参数（第3.2节），通过逗号分隔。 在一个括号 `()` 中跟随构造器参数，同样使用逗号分隔。

> The new keyword signals that a class (2.3) or an abstract (2.8) is being instantiated. It is followed by the type path (3.7) of the type which is to be instantiated. It may also list explicit type parameters (3.2) enclosed in <> and separated by comma,. After an opening parenthesis () follow the constructor arguments, again separated by comma ,, with a closing parenthesis) at the end.

```
class Main<T> {
    static public function main() {
        new Main<Int>(12, "foo");
    }

    function new(t:T, s:String) { }
}
```

main方法中我们实例化一个Main类自己的实例，有一个显式的类型参数 Int 和参数 12跟 "foo" 。就如我们看到的，语法非常类似函数的调用语法（第5.9节），通常称为构造函数调用。

> Within the main method we instantiate an instance of Main itself, with an explicit type parameter Int and the arguments 12 and "foo". As we can see, the syntax is very similar to the function call syntax (5.9) and it is common to speak of “constructor calls”.

5.13.for

Haxe不支持像 C 中传统的 for 循环。它的 for关键字期望一个开口的圆括号 (，然后一个变量标识符，之后一个关键字 in，和一个任意表达式作为迭代的集合；在闭口的圆括号) 之后，是任意的循环体表达式。

> Haxe does not support traditional for-loops known from C.Its for keyword expects an opening parenthesis (, then a variable identifier followed by the keyword in and an arbitrary expression used as iterating collection. After the closing parenthesis) follows an arbitrary loop body expression.

```
for (v in e1) e2;
```

类型工具确保 e1 的类型可以被迭代，典型的情况是是否有一个iterator方法返回一个Iterator类型，或者它本身是一个 Iterator类型。

> The typer ensures that the type of e1 can be iterated over, which is typically the case if it has an iterator method returning an Iterator, or if it is an Iterator itself.

变量 v 然后可以在循环体 e2 中使用，保存了e1集合的某个个体元素。

> Variable v is then available within loop body e2 and holds the value of the individual elements of collection e1.

Haxe有一个特别的范围操作符控制迭代次数。这是一个二院操作符，操作两个操作数：min...max，返回一个从min（包括）到max（不包括）的IntIterator类型的实例。注意，max不能小于min。

> Haxe has a special range operator to iterate over intervals. It is a binary operator taking two Int operands: min...max returns an IntIterator instance that iterates from min (inclusive) to max (exclusive). Note that max may not be smaller than min.

```
for (i in 0...10) trace(i); // 0 to 9
```

for表达式的类型总是 Void ，也就是说没有值可以被使用作为右侧表达式。

> The type of a for expression is always Void, meaning it has no value and cannot be used as right-side expression.

循环的控制流被 break（第5.20节）表达式影响，也受 continue（第5.21节）表达式影响。

> The control flow of loops can be affected by break (5.20) and continue (5.21) expressions.

5.14.while

一个普通的 while 循环由 while 关键字开始，后跟一个开口的圆括号 (，其后是条件表达式和一个闭口的圆括号) 。然后跟一个循环体表达式：

> A normal while loop starts with the while keyword, followed by an opening parenthesis (, the condition expression and a closing paranthesis). After that follows the loop body expression:

```
while(condition) expression;
```

条件表达式必须为 Bool 类型。

> The condition expression has to be of type Bool.

对应每次迭代，条件表达式都会执行。如果它执行结果为 false ，循环则停止，否则执行循环体。

> Upon each iteration, the condition expression is evaluated. If it evaluates to false, the loop stops, otherwise it evaluates the loop body expression.

```
class Main {
    static public function main() {
        var f = 0.0;
        while (f < 0.5) {
            trace(f);
            f = Math.random();
        }
    }
}
```

这类的while循环不保证一定执行循环体表达式：如果条件从开始就不匹配，它不会执行。这和 do-while循环（第5.15节）不同。

> This kind of while-loop is not guaranteed to evaluate the loop body expression at all: If the condition does not hold from the start, it is never evaluated. This is different for do-while loops (5.15).

5.15.do-while

do-while 循环由 do 关键字开始，后跟循环体表达式。其后跟while 关键字，然后是一个 开口的圆括号 (，再跟条件表达式和一个闭口的圆括号) ：

> A do-while loop starts with the do keyword followed by the loop body expression. After that follows the while keyword, an opening parenthesis (, the condition expression and a closing parenthesis):

```
do expression while(condition);
```

条件表达式必须为Bool类型。

> The condition expression has to be of type Bool.

如语法所暗示的，循环体表达式至少会执行一次，这和 while循环（第5.14节）不同。

> As the syntax suggests, the loop body expression is always evaluated at least once, unlike while (5.14) loops.

5.16.if

条件表达式由关键字 if 开始，然后条件表达式放在一个 () 内，如果条件符合，表达式将会执行：

> Conditional expressions come in the form of a leading if keyword, a condition expression enclosed in parentheses () and a expression to be evaluated in case the condition holds:

```
if (condition) expression;
```

条件表达式必须为Bool类型。

> The condition expression has to be of type Bool.

可选的，表达式可以后跟 else 关键字作为另一个表达式，如果条件不满足，可以执行else后的表达式：

> Optionally, expression may be followed by the else keyword as well as another expression to be evaluated if the condition does not hold:

```
if (condition) expression1 else expression2;
```

这里，expression2 可以认为是另一个 if 表达式：

> Here, expression2 may consist of another if expression:

```
if (condition1) expression1
else if(condition2) expression2
else expression3
```

是否一个 if 表达式的值是必须的，例如，对于 var x = if(condition) expression1 else expression2 ，类型工具确保 expression1的类型和 expression2 统一（第3.5节）。如果没有 else 表达式被给定，类型被推断为 Void 。

> If the value of an if expression is required,e.g. forvar x = if(condition) expression1 else expression2, the typer ensures that the types of expression1 and expression2 unify (3.5). If no else expression is given, the type is inferred to be Void.

5.17.switch

一个基础的 switch 表达式由 switch 关键字开始，后面是开关的主题表达式，然后是大括号内的case 表达式。case 表达式都是以case 关键字开始，后跟一个模式表达式，还有一种 defalut 关键字定义默认行为。每个case 后都跟一个冒号和一个可选的case 体表达式：

> A basic switch expression starts with the switch keyword and the switch subject expression, as well as the case expressions between curly braces {}. Case expressions either start with the case keyword and are followed by a pattern expression, or consist of the default keyword. In both cases a colon : and an optional case body expression follows:

```
switch subject {
    case pattern1: case-body-expression-1;
    case pattern2: case-body-expression-2;
    default: default-expression;
}
```

case 体表达式从不失败，所以 break（第5.20节）关键字在Haxe是不支持的。

> Case body expressions never “fall through”,sothe break (5.20) keyword is not supported in Haxe.

switch 表达式可以用作一个值；这种情况，所有case体表达式的类型和default表达式的类型必须是统一的（第3.5节）。

> Switch expressions can be used as value; in that case the types of all case body expressions and the default expression must unify (3.5).

更多关于模式匹配表达式的语法细节，在模式匹配（第6.4节）中详述。

> Further details on syntax of pattern expressions are detailed in Pattern Matching(Section6.4).

5.18.try/catch

Haxe允许捕获值，使用 try/catch 语法：

> Haxe allows catching values using its try/catch syntax:

```
try try-expr
catch(varName1:Type1) catch-expr-1
catch(varName2:Type2) catch-expr-2
```

如果在运行时try表达式引发一个 throw（第5.22节），它可以被任何后续的 catch块捕捉到。这些块由下面部分组成：

> If during runtime the evaluation of try-expression causes a throw(5.22),it can be caught by any subsequent catch block. These blocks consist of

- 一个变量名用来保存被抛出的值
- 一个显式的类型注释，决定捕捉哪种类型的值
- 这种情况下要执行的表达式

> - a variable name which holds the thrown value,
> - an explicit type annotation which determines which types of values to catch, and
> - the expression to execute in that case.

Haxe 允许抛出和捕捉任何类型的值，它不限于继承自一个特定的异常或者错误类的类型。catch块从上至下检查，第一个和抛出的值类型兼容的被采用。

> Haxe allows throwing and catching any kind of value, it is not limited to types inheriting from a specific exception or error class. Catch blocks are checked from top to bottom with the first one whose type is compatible with the thrown value being picked.

这个过程有许多和编译时的合一（第3.5节）行为的相似之处。然而，因为检查必须在运行时进行，所以有几个限制：

This process has many similarities to the compile-time unification (3.5) behavior. However, since the check has to be done at runtime there are several restrictions:

- 类型必须在运行时存在：类实例（第2.3节），enum实例（第2.4节），抽象核心类型（第2.8.7节）和 动态类型（第2.7节）。
- 类型参数必须只能为 Dynamic（第2.7节）。

> - The type must exist at runtime: Class instances (2.3), enum instances (2.4), abstract core types (2.8.7) and Dynamic (2.7).
> - Type parameters can only be Dynamic (2.7).

5.19.return

一个return表达式可以返回也可以不返回一个值表达式：

> A return expression can come with or without an value expression:

```
return;
return expression;
```

它会离开它被声明所在的控制流最深的函数，当局部函数（第5.11节）被调用时需要被区别：

> It leaves the control-flow of the innermost function it is declared in, which has to be distinguished when local functions (5.11) are involved:

```
function f1() {
    function f2() {
        return;
    }
    f2();
    expression;
}
```

return会离开局部函数 f2，但不会离开 f 1，意味着 expression仍然会被执行。

> The return leaves local function f2, but not f1, meaning expression is still evaluated.

如果 return 不带值表达式使用，类型工具确保函数返回类型会返回Void 。如果它有一个值表达式，类型工具 统一（第3.5节）它的类型为它所返回的函数的返回类型（显式指定或者通过前面的return表达式推断）。

> If return is used without a value expression, the typer ensures that the return type of the function it returns from is of Void. If it has a value expression,the typer unifies(3.5) its type with the return type (explicitly given or inferred by previous return expressions) of the function it returns from.

5.20.break

break关键字会离开它所声明的控制流最深的循环，停止进一步的迭代：

> The break keyword leaves the control flow of the innermost loop (for or while) it is declared in, stopping further iterations:

```
while(true) {
    expression1;
    if (condition) break;
    expression2;
}
```

这里，expression1每个迭代都会被执行，但是当条件满足，expression2则不再执行。

> Here,expression1 is evaluated for each iteration,but as soon as condition holds,expression2 is not evaluated anymore.

类型工具确保它只出现在一个循环中。break关键字在Haxe并不支持用于 switch case表达式（第5.17节）。

> The typer ensures that it appears only within a loop. The break keyword in switch cases (5.17) is not supported in Haxe.

5.21.continue

continue关键字结束它所声明的最深处的当前迭代，使循环条件被检查于下一次迭代：

> The continue keyword ends the current iteration of the innermost loop (for or while) it is declared in, causing the loop condition to be checked for the next iteration:

```
while(true) {
    expression1;
    if(condition) continue;
    expression2;
}
```

这里，expression1每次迭代都被执行，但是如果条件满足，expression2不再为当前迭代执行。不像break，迭代仍然继续。

> Here, expression1 is evaluatedfor each iteration, but if condition holds, expression2 is not evaluated for the current iteration. Unlike break, iterations continue.

类型工具确保它只出现在循环中。

> The typer ensures that it appears only within a loop.

5.22.throw

Haxe允许抛出一个类型的值，使用 throw 语法：

> Haxe allows throwing any kind of value using its throw syntax:

```
throw expr
```

被抛出的值可以被catch块（第5.18节）捕捉。如果没有这样的块捕捉，行为取决于目标平台。

> A value which is thrown like this can be caught by catch blocks (5.18). If no such block catches it, the behavior is target-dependent.

5.23.类型转换

Haxe允许两种类型转换：

> Haxe allows two kinds of casts:

```
cast expr; // 不安全的类型转换
cast (expr, Type); // 安全转换
```

5.23.1.不安全转换

Unsafe casts are useful to subvert the type system. The compiler types expr as usual and then wraps it in a monomorph (2.9). This allows the expression to be assigned to anything. 不安全的类型转换不使用任何dynamic（第2.7节）类型，如下面例子展示的：

> Unsafe casts do not introduce any dynamic (2.7) types, as the following example shows:

```
class Main {
    public static function main() {
        var i = 1;
        $type(i); // Int
        var s = cast i;
        $type(s); // Unknown<0>
        Std.parseInt(s);
        $type(s); // String
    }
}
```

变量 i 类型化为 Int，然后经过不安全转换 cast i 之后赋值到变量。这使 s 成为一个 unknown 类型，一个单形。根据合一（第3.5节）的一般规则，它可以之后被绑定为任何类型，例如例子中的 String 。

> Variable i is typed as Int and then assigned to variables using the unsafe cast cast i. This causes s to be of an unknown type,amonomorph. Following the usual rules of unification(3.5), it can then be bound to any type, such as String in this example.

这些转换被称为不安全的，因为无效转换的运行时行为没有被定义。而多数 动态目标语言（第2.2节）可能可以工作，但是在静态目标语言（第2.2节）可能导致未定义的错误。

> These casts are called ”unsafe” because the runtime behavior for invalid casts is not defined. While most dynamic targets (2.2) are likely to work, it might lead to undefined errors on static targets (2.2).

不安全转换几乎没有运行时的开销。

> Unsafe casts have little to no runtime overhead.

5.23.2.安全转换

和不安全转换（第5.23.1节）不同，一个失败的转换的运行时行为在安全转换中被定义：

> Unlike unsafe casts(5.23.1),the runtime behavior in case of a failing cast is defined for safe casts:

```
class Base {
    public function new() { }
}

class Child1 extends Base {}

class Child2 extends Base {}

class Main {
    public static function main() {
        var child1:Base = new Child1();
        var child2:Base = new Child2();
        cast(child1, Base);
        // Exception: Class cast error
        cast(child1, Child2);
    }
}
```

在这个例子中，我们首先转换一个Child1类的实例为Base，因为Child1是一个Base的子类（第2.3.2节），所以转换成功。然后尝试转换同样的类实例为 Child2，这是不被允许的，因为Child2的实例并不是Child1类型的。

> In this example we first cast a class instance of type Child1 to Base,which succeeds because Child1 is a child class (2.3.2) of Base. We then try to cast the same class instance to Child2, which is not allowed because instances of Child2 are not instances of Child1.

Haxe编译器保证一个String类型的异常，在这种情况被抛出（第5.22节）。这个异常可以使用 try/catch 块捕捉到。

> The Haxe compiler guarantees that an exception of type String is thrown(5.22) in this case. This exception can be caught using a try/catch block (5.18).

安全转换有一个运行时的开销。理解编译器已经发生了类型检查是很重要的，所以添加手动的检查是冗余的，例如，使用 Std.is 。预期的使用是try安全转换，然后捕捉String类型的异常。

> Safe casts have a runtime overhead. It is important to understand that the compiler already generates type checks,so it is redundant to add manual checks,e.g. usingStd.is. The intended usage is to try the safe cast and catch the String exception.

5.24.类型检查

从**Haxe 3.1.0**以后

> **Since Haxe 3.1.0**

通过下面的语法，可以使用编译时类型检查：

> It is possible to employ compile-time type checks using the following syntax:

```
(expr : type)
```

括号是强制性的。不想安全转换（第5.23.2），这个结构没有运行时的影响。它有两个编译时的影响：

> The parentheses are mandatory. Unlike safe casts (5.23.2) this construct has no run-time impact. It has two compile-time implications:

- 从上至下的推断（第3.6.1节）用于类型化 expr 为 type 类型
- 结果的类型化的表达式统一（第3.5节）为type 类型

> - Top-down inference (3.6.1) is used to type expr with type type.
> - The resulting typed expression is unified (3.5) with type type.

这通常影响两个操作，如当执行绝对的标识符解析（第3.7.3节）时给定类型被用为预期的类型，和抽象类型的转换的合一检查。

> This has the usual effect of both operations such as the given type being used as expected type when performing unqualified identifier resolution (3.7.3) and the unification checking for abstract casts (2.8.1).

6.语言特性

抽象类型（第2.8节）： 一个抽象类型是一个编译时构造，在运行时以一个不同的方式表示。这允许给存在的类型一个全新的意义。

> Abstract types (2.8): An abstract type is a compile-time construct which is represented in a different way at runtime. This allows giving a whole new meaning to existing types.

外部类（第6.2节）： 外部类可以被用于以一个类型安全的方式描述目标语言特定的交互。

> Extern classes (6.2): Externs can be used to describe target-specific interaction in a type-safe manner.

匿名结构（第2.5节）： 数据可以被简单的组织为匿名结构，减少小型数据类的必要性。

> Anonymous structures (2.5): Data can easily be grouped in anonymous structures, minimizing the necessity of small data classes.

```
var point = { x: 0, y: 10 };
point.x += 10;
```

数组推导（第6.6节）： 使用 for 循环和一些逻辑快速创建和填充数组。

> Array Comprehension (6.6): Create and populate arrays quickly using for loops and logic.

```
var evenNumbers = [ for (i in 0...100) if (i\%2==0) i ];
```

类，接口和继承（第2.3节）： Haxe允许用类组织代码，使其成为一个面向对象语言。通常相关的功能如Jave等语言所支持的，包括继承和接口。

> Classes, interfaces and inheritance (2.3): Haxe allows structuring code in classes, making it an object-oriented language. Common related features known from languages such as Java are supported, including inheritance and interfaces.

条件编译（第6.1节）： 条件编译允许根据编译参数编译特定的代码。这有助于抽象目标语言特定的差异，但是也可以用于其他的目的，如更详细的调试。

> Conditional compilation (6.1): Conditional Compilation allows compiling specific code depending on compilation parameters. This is instrumental for abstracting target-specific differences,but can also be used for other purposes, such as more detailed debugging.

```
\#if js
    js.Browser.alert("Hello");
\#elseif sys
    Sys.println("Hello");
\#end
```

（广义的）代数数据类型（第2.4节）： 结构可以通过代数数据类型（ADT）描述，如Haxe语言中的枚举。除此之外，Haxe支持它们的广义的变体如GADT。

> (Generalized) Algebraic Data Types (2.4): Structure can be expressed through algebraic data types (ADT), which are known as enums in the Haxe Language. Furthermore, Haxe supports their generalized variant known as GADT.

```
enum Result {
    Success(data:Array<Int>);
    UserError(msg:String);
    SystemError(msg:String, position:PosInfos);
}
```

内联调用（第4.4.2节）： 函数可以被设计为内联，使它们的代码直接插入调用的位置。通过手动的内联不用使代码重复这可以产生显著的效能提升。

> Inlined calls (4.4.2): Functions can be designated as being inline, allowing their code to be inserted at call-site. This can yield significant performance benefits with out resorting to code duplication via manual inlining.

迭代器（第6.7节）： 迭代一组值，例如一个数组的元素，在Haxe中可以很容易的迭代。定制类可以快速的实现迭代器功能来允许迭代。

> Iterators (6.7): Iterating over a set of values, e.g. the elements of an array, is very easy in Haxe courtesy of iterators. Custom classes can quickly implement iterator functionality to allow iteration.

```
for (i in [1, 2, 3]) {
    trace(i);
}
```

局部函数和闭包（第5.11节）： Haxe中的函数不限于类字段，并可以被声明为表达式，允许强大的闭包。

> Local functions and closures (5.11): Functions in Haxe are not limited to class fields and can be declared in expressions as well, allowing powerful closures.

```
var buffer = "";
function append(s:String) {
    buffer += s;
}
append("foo");
append("bar");
trace(buffer); // foobar
```

元数据（第6.9节）： 添加元数据到字段，类或者表达式。这可以和编译器、宏，或者运行时的类沟通信息。

> Metadata (6.9): Add metadata to fields, classes or expressions. This can communicate information to the compiler, macros, or runtime classes.

```
class MyClass {
    @range(1, 8) var value:Int;
}
trace(haxe.rtti.Meta.getFields(MyClass).value.range); // [1,8]
```

静态扩展（第6.3节）： 存在的类和其它类型可以被额外的功能来扩展，通过使用静态扩展。

> Static Extensions (6.3): Existing classes and other types can be augmented with additional functionality through using static extensions.

```
 using StringTools;
 " Me & You ".trim().htmlEscape();
```

字符串插值（第6.5节）： 字符串通过一个单引号声明，可以在当前的上下文访问变量。

> String Interpolation (6.5): Strings declared with a single quotes are able to access variables in the current context.

```
trace('My name is $name and I work in ${job.industry}');
```

偏函数应用（第6.8节）： 任何函数可以应用为局部的，提供某些参数的值，然后保留其它的作为之后的字段。

> Partial function application (6.8): Any function can be applied partially, providing the values of some arguments and leaving the rest to be filled in later.

```
var map = new haxe.ds.IntMap();
var setToTwelve = map.set.bind(_, 12);
setToTwelve(1);
setToTwelve(2);
```

模式匹配（第6.4节）： 复杂的结构可以被根据模式来匹配，从一个枚举或者一个结构中提取信息，并对特定的值组合定义特定的操作。

> Pattern Matching (6.4): Complex structures can be matched against patterns, extracting information from an enum or a structure and defining specific operations for specific value combination.

```
var a = { foo: 12 };
switch (a) {
    case { foo: i }: trace(i);
    default:
}
```

属性（第4.2节）： 变量类字段可以被涉及为属性，通过定制的read和write访问，可以更精细的访问控制。

> Properties (4.2): Variable class fields can be designed as properties with custom read and write access, allowing fine grained access control.

```
public var color(get,set);
function get_color() {
    return element.style.backgroundColor;
}
function set_color(c:String) {
    trace('Setting background of element to $c');
    return element.style.backgroundColor = c;
}
```

访问控制（第6.10节）： 访问控制语言特性使用Haxe元数据语法来禁止或者允许访问类或者字段。

> Access control (6.10): The access control language feature uses the Haxe metadata syntax to force or allow access classes or fields.

类型参数、约束和变异（第3.2节）： 类型可以通过类型参数来参数化，使类型化的容器和其它复杂的数据结构可用。类型参数也可以被约束为某些类型并遵守变异规则。

> Type Parameters, Constraints and Variance (3.2): Types can be parametrized with type parameters, allowing typed containers and other complex data structures. Type parameters can also be constrained to certain types and respect variance rules.

```
class Main<A> {
   static function main() {
       new Main<String>("foo");
```

```
        new Main(12); // use type inference
    }

    function new(a:A) { }
}
```

6.1.条件编译

Haxe允许通过使用 #if 、#elseif 和 #else 进行条件编译，并检查编译器标记。

> Haxe allows conditional compilation by using #if, #elseif and #else and checking for compiler flags.
>
> 定义：编译器标记 一个编译器标记是一个配置值，可以改变编译过程。可以调用命令行使用 -D key=value 或者 只有 -D key的形式对标记进行设置，只有key的时候值默认为 1 。编译器也设置了一些内部标记在不同的编译步骤之间传递信息。 [warning] **Definition**: Compiler Flag A compiler flag is a configurable value which may influence the compilation process. Such a flag can be set by invoking the command line with -D key=value or just -D key, in which case the value defaults to "1". The compiler also sets several flags internally to pass information between different compilation steps.

这个例子展示了条件编译的用法：

> This example demonstrates usage of conditional compilation:

```
class Main {
    public static function main(){
        #if !debug
            trace("ok");
        #elseif (debug_level > 3)
            trace(3);
        #else
            trace("debug level too low");
        #end
    }
}
```

不使用任何标记编译只保留 trace("ok"); 这行。另一个分支在解析文件时被丢弃。这些其他的分支必须包含有效的Haxe语法，但是代码不会执行类型检查。

> Compiling this without any flags will leave only the trace("ok"); line in the body of the main method. The other branches are discarded while parsing the file. These other branches must still contain valid Haxe syntax, but the code is not type-checked.

#if 之后的条件和 #elseif之后的条件允许如下的表达式：

> The conditions after #if and #elseif allow the following expressions:

- 任何标识符被编译器标记替换为同名的。注意 -D -some-flag 指示标记 some-flag，somin_flag被定义。
- String，Int和Float常量值被直接使用。
- 布尔操作符 &&， || ，和 ! 如同预期那样工作，然而整个表达式必须被完全放在括号中。
- 操作符 ==，!=，>， >=，<，<=可以用来对比值。
- 括号可以用来像往常一样组织表达式。

> - Any identifier is replaced by the value of the compiler flag by the same name. Note that -D some-flag from command line leads to the flags some-flag and some_flag to be defined.
> - The values of String, Int and Float constants are used directly.
> - The boolean operators && (and), || (or) and ! (not) work as expected, however the full expression must be completely contained by parentheses.
> - The operators ==, !=, >, >=, <, <= can be used to compare values.
> - Parentheses () can be used to group expressions as usual.

Haxe解析器不解析 some-flag为一个字符串令牌，而是将她读为一个减法二院操作符 some - flag 。这种情况，下划线的版本 some_flag 就有使用的必要了。

> The Haxe parser does not parse some-flag as a single token and instead reads it as a subtraction binary operator some - flag. In cases like this the underscore version some_flag has to be used.

内建编译器标记 一个所有内部定义的详尽列表可以通过在Haxe编译器 调用 --help-defines 参数来获得。Haxe编译器在每次编译允许多个 -D 标记。

> **Built-in Compiler Flags** An exhaustive list of all built-in defines can be obtained by invoking the Haxe Compiler with the --help-defines argument. The Haxe Compiler allows multiple -D flags per compilation.

也可以查看编译器标记列表（第7.1节）。

> See also the Compiler Flags list (7.1).

6.2.Externs

Extren可以用一个类型安全的方式来描述目标语言特定的交互。它们像普通类一样定义，除了：

> Externs can be used to describe target-specific interaction in a type-safe manner. They are defined like normal classes, except that

- class关键字被extern 关键字领先，
- 方法（第4.3节）没有表达式，
- 所有的参数和返回类型需要是显式的。

> - the class keyword is preceded by the extern keyword,
> - methods (4.3) have no expressions and
> - all argument and return types are explicit.

常见的例子在Haxe标准库（第10章），Math 类，摘录如下：

> A common example from the Haxe Standard Library (10) is the Math class, as an excerpt shows:

```
extern class Math
{
    static var PI(default,null) : Float;
    static function floor(v:Float):Int;
}
```

我们看到，externs 既可以定义方法也可以定义变量（时尚，PI是声明为一个只读属性（第4.2节））。一旦这个信息对于编译器可用，它启用相应的字段访问，也被称为类型：

> We see that externs can define both methods and variables (actually, PI is declared as a readonly property (4.2)). Once this information is available to the compiler, it allows field access accordingly and also knows the types:

```
class Main {
    static public function main() {
        var pi = Math.floor(Math.PI);
        $type(pi); // Int
    }
}
```

这这可以运行，因为floor方法返回类型声明为Int。

> This works because the return type of method floor is declared to be Int.

Haxe标准库带有许多外部类，对于Flash和JavaScript目标语言。它们使可以以类型安全的方式访问原生的APIs，当作设计高层APIs的工具。Externs也可以应用haxelib（第11章）中许多流行的原生库。

> The Haxe Standard Library comes with many externs for the Flash and JavaScript target. They allow accessing the native APIs in a type-safe manner and are instrumental for designing higher-level APIs. There are also externs for many popular native libraries on haxelib (11).

Flash，Java和C#目标语言允许通过命令行（第7章）直接包括原生的库。目标特定的细节在每个目标语言细节（第12章）等各自的章节。

> The Flash, Java and C# targets allow direct inclusion of native libraries from command line (7). Target-specific details are explained in the respective sections of Target Details (Chapter 12).

一些目标语言如Python或者JavaScript可能需要产生附加的 import 代码，为原生模块加载一个extern 类。Haxe提供声明这样的依赖关系在各自的目标语言向细节（第12章）的方式。。

> Some targets such as Python or JavaScript may require generating additional ”import” code that loads an extern class from a native module. Haxe provides ways to declare such dependencies also described in respective sections Target Details (Chapter 12).

Haxe3.2.0以后剩余的参数和类型选择

> Rest arguments and type choices Since Haxe 3.2.0

haxe.extern 包提供两种类型帮助映射原生的语义到Haxe：

> The haxe.extern package provides two types that help mapping native semantics to Haxe:

Rest 这个类型可以被实用为一个最终的函数参数，来使传递一个额外的调用参数任意的数值到函数。类型参数可以被使用来限制这些参数为某个特定类型。

> **Rest**: This type can be used as a final function argument to allow passing an arbitrary number of additional call arguments. The type parameter can be used to constrain these arguments to a specific type.

EitherType：这个类型允许使用任何一个类型参数类型，因此表示了一个类型选择。它可以被嵌套来允许更多的类型。

> **EitherType**: This type allows using either of its parameter types,thus representing a type choice. It can be nested to allow more than two different types.

我们在下面代码示例中演示了其用法：

> We demonstrate the usage in this code sample:

```
import haxe.extern.Rest;
import haxe.extern.EitherType;

extern class MyExtern {
    static function f1(s:String, r:Rest<Int>):Void;
    static function f2(e:EitherType<Int, String>):Void;
}

class Main {
    static function main() {
        MyExtern.f1("foo", 1, 2, 3); // use 1, 2, 3 as rest argument
        MyExtern.f1("foo"); // no rest argument
        //MyExtern.f1("foo", "bar"); // String should be Int

        MyExtern.f2("foo");
        MyExtern.f2(12);
        //MyExtern.f2(true); // Bool should be EitherType<Int, String>
    }
}
```

6.3.静态扩展

> 定义：静态扩展 一个静态扩展允许伪装扩展已存在的类型而不用修改它们的源码。在Haxe中这是通过声明一个静态方法，第一个参数是要扩展的类型，然后带入定义类到上下文中就可以使用。 [warning] Definition: Static Extension A static extension allows pseudo-extending existing types without modifying their source. In Haxe this is achieved by declaring a static method with a first argument of the extending type and then bringing the defining class into context through using.

静态扩展是一个强大的工具，使得可以不用实际修改就对类型进行扩展。下面的示例展示用法：

> Static extensions can be a powerful tool which allows augmenting types without actually changing them. The following example demonstrates the usage:

```
using Main.IntExtender;

class IntExtender {
    static public function triple(i:Int) {
        return i * 3;
    }
}

class Main {
    static public function main() {
        trace(12.triple());
    }
}
```

显然，Int并没有原生提供一个 triple 方法，然而这个程序编译并输出预期的 36 。这是因为调用 12.triple() 是转换到 IntExtender.triple(12) 。这有三个要求：

> Clearly,Int does not natively provide a triple method,yet this program compiles and outputs 36 as expected. This is because the call to 12.triple() is transformed into IntExtender.triple(12). There are three requirements for this:

1. 字面的值12和第一个参数都是已知的Int类型
2. IntExtender类通过 using Main.IntExtender 带入到上下文
3. Int本身没有 triple字段（如果它有，那个字段会优先于静态扩展）

> 1. Both the literal 12 and the first argument of triple are known to be of type Int.
> 2. The class IntExtender is brought into context through using Main.IntExtender.
> 3. Int does not havea triple field by itself (if it had,that field would take priority over the static extension).

静态扩展通常认为是语法糖，事实也是如此，但是值得注意的是，它们可以对代码可读性有一个戏剧性影响：替代嵌套的调用形式如f1(f2(f3(f4(x)))),使用链式调用形式如x.f4().f3().f2().f1() 。

> Static extensions are usually considered syntactic sugar and indeed they are, but it is worth noting that they can have a dramatic effect on code readability: Instead of nested calls in the form of f1(f2(f3(f4(x)))), chained calls in the form of x.f4().f3().f2().f1() can be used.

遵守前面讲述过的解析顺序（第3.7.3节），多个使用的表达式从底部到头部检查，而在每个模块的类型以及在每个类型中的字段从头至尾检查。使用一个模块（而不是一个模块中的特定类型，查看模块和路径（第3.7节））作为静态扩展会把所有它的类型带入上下文中。

> Following the rules previously described in Resolution Order(Section3.7.3),multiple using expressions are checked from bottom to top, with the types within each module as well as the fields within each type being checked from top to bottom. Using a module (as opposed to a specific type of a module, see Modules and Paths (Section 3.7)) as static extension brings all its types into context.

6.3.1.标准库中的静态扩展

在Haxe标准库中的一些类适用于静态扩展用法。下面的例子展示了StringTools的用法：

> Several classes in the Haxe Standard Library are suitable for static extension usage. The next example shows the usage of StringTools:

```
using StringTools;

class Main {
    static public function main() {
        "adc".replace("d", "b");
    }
}
```

String本身并没有替换功能， using StringTools 静态扩展提供了一个。通常，JavaScript输出很好的显式了这个转换：

> While String does not have a replace functionality by itself, the using StringTools static extension provides one. As usual, the JavaScript output nicely shows the transformation:

```
Main.main = function() {
    StringTools.replace("adc","d","b");
}
```

如下Haxe标准库中的类都是设计作为静态扩展使用：

> The following classes from the Haxe Standard Library are designed to be used as static extensions:

StringTools：提供字符串的扩展功能，例如替换和去除空格。 **Lambda**：提供功能方法到可迭代对象。 **haxe.EnumTools**：提供类型信息功能到enum和它们的实例。 **haxe.macro.Tools**：为和宏有关的操作提供不同的扩展（查看 工具（第9.4节））

> **StringTools**: Provides extended functionality on strings, such as replacing or trimming. **Lambda**: Provides functional methods on iterables. **haxe.EnumTools**: Provides type information functionality on enums and their instances. **haxe.macro.Tools**: Provides different extensions for working with macros (see Tools (Section 9.4)).

使用**using**

> 花絮：使用using 自从using关键字被添加到语言中，讨论某些 使用using的问题 或者 using的影响变得非常常见。在很多情况下的英语变得尴尬，所以手册的作者决定把这个使用实际的功能称呼：静态扩展。
> [warning] **Trivia**: “using” using Since the using keyword was added to the language, it has been common to talk about certain problems with “using using” or the effect of “using using”. This makes for awkward English in many cases, so the author of this manual decided to call the feature by what it actually is: Static extension.

大意可能就是说，**using**是作为静态扩展的关键字存在。如果本节标题使用 **using using** 会造成混淆。

6.4.模式匹配

本节内容：

6.4.1.介绍

模式匹配是根据一个匹配的指定值进行分支处理，尽可能深的模式。在Haxe中，所有的模式匹配通过 switch表达式（第5.17节）处理，在其中通过个体的case语句表示模式。这里我们将探索不同的模式的语法，使用这个数据结构作为运行例子：

> Pattern matching is the process of branching depending on a value matching given, possibly deep patterns. In Haxe, all pattern matching is done within a switch expression (5.17) where the individual case expressions represent the patterns. Here we will explore the syntax for different patterns using this data structure as running example:

```
enum Tree<T> {
    Leaf(v:T);
    Node(l:Tree<T>, r:Tree<T>);
}
```

一些模式匹配器基础包括:

> Some pattern matcher basics include:

- 模式总是从头至尾匹配
- 最上面的匹配了输入的值的模式，它的表达式将被执行
- 一个 匹配任何，所以 *case* : 是等同于 default: 的

> - Patterns will always be matched from top to bottom.
> - The topmost pattern that matches the input value has its expression executed.
> - A *pattern matches anything, so case* : is equal to default:

6.4.2.枚举匹配

枚举可以以一个自然的方式通过它们的构造器进行匹配：

> Enums can be matched by their constructors in a natural way:

```
var myTree = Node(Leaf("foo"), Node(Leaf("bar"), Leaf("foobar")));
var match = switch(myTree) {
   // matches any Leaf
   case Leaf(_): "0";
   // matches any Node that has r = Leaf
   case Node(_, Leaf(_)): "1";
   // matches any Node that has has
   // r = another Node, which has
   // l = Leaf("bar")
   case Node(_, Node(Leaf("bar"), _)): "2";
   // matches anything 12 case _: "3";
}
trace(match); // 2
```

模式匹配器会检查每个 case ，从头至尾，采用第一个匹配输入值的 case 。下面关于case规则的说明解释帮助你理解该过程：

> The pattern matcher will check each case from top to bottom and pick the first one that matches the input value. The following manual interpretation of each case rule helps understanding the process:

case Leaf(*)*：匹配失败，因为 *myTree* 是一个*i*额 *Node case Node(*,Leaf(*))*：匹配失败，因为*myTree*右侧的子树不是一个*Leaf*，而是另一个 *Node* ***case Node(*,Node(Leaf("bar"),_))**：匹配成功

case _：不被检查，因为前一行已经匹配成功

> **case Leaf(_)**: matching fails because myTree is a Node **case Node(*, Leaf()*)**: matching fails because the right sub-tree of myTree is not a Leaf, but another Node **case Node(*, Node(Leaf("bar"),*))**: matching succeeds **case _**: this is not checked here because the previous line matched

6.4.3.变量捕获

可以捕获一个子模式的任何值，通过匹配它对应的一个标识符：

> It is possible to catch any value of a sub-pattern by matching it against an identifier:

```
var myTree = Node(Leaf("foo"), Node(Leaf("bar"), Leaf("foobar")));
var name = switch(myTree) {
    case Leaf(s): s;
    case Node(Leaf(s), _): s;
    case _: "none";
}
trace(name); // foo
```

这会返回如下之一：

> This would return one of the following:

- 如果 myTree 是一个 Leaf，它的name 被返回
- 如果myTree 是一个 Node，并且左侧的子树是一个 Leaf，它的name被返回（这应用在这里，返回“foo”）
- 否则 “none”被返回

> - If myTree is a Leaf, its name is returned.
> - If myTree is a Node whose left sub-tree is a Leaf,its name is returned (this will apply here, returning "foo").

> - Otherwise "none" is returned.

还可以使用 = 来捕获值，这更进一步的匹配：

> It is also possible to use = to capture values which are further matched:

```
var node = switch(myTree) {
    case Node(leafNode = Leaf("foo"), _): leafNode;
    case x: x;
}
trace(node); // Leaf(foo)
```

这里，如果输入匹配 leafNode 是绑定到 Leaf("foo") 。在所有其他情况，myTree 本身是被返回：case x 工作方式类似于 case _ ，它匹配任何，但是使用一个i额标识名称如 x ，它也会绑定匹配的值到这个变量。

> Here, leafNode is bound to Leaf("foo") if the input matches that. In all other cases, myTree itself is returned: case x works similar to case _ in that it matches anything, but with an identifier name like x it also binds the matched value to that variable.

6.4.4.结构匹配

还可以根据匿名结构的字段和实例进行匹配：

> It is also possible to match against the fields of anonymous structures and instances:

```
var myStructure = {
    name: "haxe",
    rating: "awesome"
};
var value = switch(myStructure) {
    case { name: "haxe", rating: "poor" }:
        throw false;
    case { rating: "awesome", name: n }:
        n;
    case _:
        "no awesome language found";
}
trace(value); // haxe
```

在第二种情况，如果评定匹配 “awesome”我们绑定匹配的 name 字段 为标识符 n，。当然这个结构也可以被放入前一个例子中的 Tree 来联合结构和枚举匹配。

> In the second case we bind the matched name field to identifier n if rating matches "awesome". Of course this structure could also be put into the Tree from the previous example to combine structure and enum matching.

类实例的一个限制是，不能匹配它们父类的字段。

> A limitation with regards to class instances is that you cannot match against fields of their parent class.

6.4.5.数组匹配

数组可以使用固定的长度匹配：

> Arrays can be matched on fixed length:

```
var myArray = [1, 6];
var match = switch(myArray) {
    case [2, _]: "0";
    case [_, 6]: "1";
    case []: "2";
    case [_, _, _]: "3";
    case _: "4";
}
trace(match); // 1
```

这会输出1，因为 array[1]匹配6，array[0]允许是任何内容。

> This will trace 1 because array[1] matches 6, and array[0] is allowed to be anything.

6.4.6.Or 模式

| 操作符用来在模式中描述多个接受的模式：

> The | operator can be used anywhere within patterns to describe multiple accepted patterns:

```
var match = switch(7) {
    case 4 | 1: "0";
    case 6 | 7: "1";
    case _: "2";
}
trace(match); // 1
```

在一个 or 模式中，如果有一个捕获的变量，它必须也出现在它的子模式中。

> If there is a captured variable in an or-pattern, it must appear in both its sub-patterns.

6.4.7.守护(Guard)

也可以更进一步的限制模式，通过使用 `case ... if(condition):` 语法：

> It is also possible to further restrict patterns with the case ... if(condition): syntax:

```
var myArray = [7, 6];
var s = switch(myArray) {
    case [a, b] if (b > a):
        b + ">" +a;
    case [a, b]:
        b + "<=" +a;
    case _: "found something else";
}
trace(s); // 6<=7
```

第一个情况有一个附加的守护条件 if(b>a) 。只有条件满足case才被采用，否则匹配继续下一个case 。

> The first case has an additional guard condition if (b > a). It will only be selected if that condition holds, otherwise matching continues with the next case.

6.4.8.多个值的匹配

数组语法可以用来匹配多个值：

> Array syntax can be used to match on multiple values:

```
var s = switch [1, false, "foo"] {
    case [1, false, "bar"]: "0";
    case [_, true, _]: "1";
    case [_, false, _]: "2";
}
trace(s); // 2
```

这和数组匹配非常相似，但是有以下不同：

> This is quite similar to usual array matching, but there are differences:

- 元素的量是固定的，所以不接受不同长度的数组
- 不能捕获switch的值到一个变量，如 case x 是不被允许的（case _ 也是）。

> - The number of elements is fixed, so patterns of different array length will not be accepted.
> - It is not possible to capture the switch value in a variable,i.e. case x isnotallowed(case _ still is).

6.4.9.提取器

Haxe 3.1.0以后

> **Since Haxe 3.1.0**

提取器允许应用变换到被匹配的值。这才需要对一个匹配的值在匹配继续之前做一个小的操作时经常用到：

> Extractors allow applying transformations to values being matched. This is often useful when a small operation is required on a matched value before matching can continue:

```
enum Test {
    TString(s:String);
    TInt(i:Int);
}

class Main {
    static public function main() {
        var e = TString("fOo");
        switch(e) {
            case TString(temp):
                switch(temp.toLowerCase()) {
                    case "foo": true;
                    case _: false;
                }
            case _: false;
        }
    }
}
```

这里我们需要TString 枚举构造器的参数值到一个变量 temp，并使用一个嵌套的 switch到temp.toLowerCase() 。很明显，我们想要匹配如果TString保存一个无论大小写的“foo”则成功。这可以通过提取器简单实现：

> Here we have to capture the argument value of the TString enum constructor in a variable temp and use a nested switch on temp.toLowerCase(). Obviously, we want matching to succeed if TString holds a value of "foo" regardless of its casing. This can be simplified with extractors:

```
enum Test {
    TString(s:String);
    TInt(i:Int);
}

class Main {
    static public function main() {
        var e = TString("fOo");
        var success = switch(e) {
            case TString(_.toLowerCase() => "foo"):
                true;
            case _:
                false;
        }
    }
}
```

提取器通过extractorExpression => match 表达式识别。编译器生成的代码类似于前面的例子，但是原生的语法被大大精简。提取器由两部分组成，被 => 操作符分隔：

> Extractors are identified by the extractorExpression => match expression. The compiler generates code which is similar to the previous example,but the original syntax was greatly simplified. Extractors consist of two parts, which are separated by the => operator:

- 左侧可以使任何的表达式，中间出现的所有下划线被替换为当前匹配的值。
- 右侧是一个模式，匹配的左侧执行的结果。

> - The left side can be any expression,where all occurrences of underscore_are replaced with the currently matched value.
> - The right side is a pattern which is matched against the result of the evaluation of the left side.

因为右侧是模式，它可以包含另外的提取器。下面的例子链接了两个提取器：

> Since the right side is a pattern, it can contain another extractor. The following example “chains” two extractors:

```
class Main {
    static public function main() {
        switch(3) {
            case add(_, 1) => mul(_, 3) => a:
                trace(a);
        }
    }

    static function add(i1:Int, i2:Int) {
        return i1 + i2;
    }

    static function mul(i1:Int, i2:Int) {
        return i1 * i2;
    }
}
```

这里输出 12 作为调用 add(3,1)的结果，3是匹配的值，mul(4,3)中4是add调用的结果。需要注意的是，a在第二个=>操作符的右侧是一个捕获的变量（第6.4.3节）。

> This traces 12 as a result of the calls to add(3, 1), where 3 is the matched value, and mul(4, 3) where 4 is the result of the add call. It is worth noting that the a on the right side of the second => operator is a capture variable (6.4.3).

当前不能在 or 模式中使用提取器：

> It is currently not possible to use extractors within or-patterns (6.4.6):

```
class Main {
    static public function main() {
        switch("foo") {
            // Extractors in or patterns are not allowed
            case (_.toLowerCase() => "foo") | "bar":
        }
    }
}
```

然而，可以把or模式用在提取器的右侧，所以前面的例子会编译而没有括号。

> However, it is possible to have or-patterns on the right side of an extractor, so the previous example would compile without the parentheses.

6.4.10.穷尽性检查

编译器确保没有被遗漏的可能的case：

> The compiler ensures that no possible cases are forgotten:

```
switch(true) {
    case false:
} // Unmatched patterns: true
```

匹配的类型 Bool 认可两个值 true 和false ，但是只有false被检查。

> The matched type Bool admits two values true and false, but only false is checked.

6.4.11.无效的模式检查

以类似的方式，编译器检测不会匹配输入的值的模式：

> In a similar fashion, the compiler detects patterns which will never match the input value:

```
switch(Leaf("foo")) {
    case Leaf(_)
        | Leaf("foo"): // This pattern is unused
    case Node(l,r):
    case _: // This pattern is unused
}
```

6.5.字符串插值

使用Haxe 3 ，通过字符串插值，不再需要手动连接字符串的各部分。一个特定的标识符，通过美元符号 $ 在一个单引号括起来的字符串中，像连接标识符一样被执行：

> With Haxe 3 it is no longer necessary to manually concatenate parts of a string due to the introduction of String Interpolation. Special identifiers, denoted by the dollar sign $ within a String enclosed by single-quote ’ characters, are evaluated as if they were concatenated identifiers:

```
var x = 12;
// The value of x is 12
trace('The value of x is $x');
```

此外，可以包括完整的表达式到字符串中，通过使用 ${expr} ，expr 是任何有效的Haxe 表达式：

> Furthermore, it is possible to include whole expressions in the string by using ${expr}, with expr being any valid Haxe expression:

```
var x = 12;
// The sum of 12 and 3 is 15
trace('The sum of $x and 3 is ${x + 3}');
```

字符串插值是一个编译时功能，不会对运行时性能产生影响。上面的例子和手动连接是等效的，和手动连接编译器生成相同的内容：

> String interpolation is a compile-time feature and has no impact on the runtime. The above example is equivalent to manual concatenation, which is exactly what the compiler generates:

```
trace("The sum of " + x +
    " and 3 is " + (x + 3));
```

当然，使用单引号括起来的字符串即使没有任何插值仍然是有效的，但是注意美元符号因为它会触发插值。如果一个实际的美元符号被使用，可以使用 $$ 双美元符号代替。

> Of course the use of single-quote enclosed strings without any interpolation remains valid, but care has to be taken regarding the $ character as it triggers interpolation. If an actual dollar-sign should be used in the string, $$ can be used.

Haxe3之前的字符串插值

> 花絮： Haxe3之前的字符串插值 从Haxe 2.09之后字符串插值就作为一个Haxe功能被引入。在那之前，必须使用宏 Std.format，跟新的字符串插值语法相比，即慢而且不够灵活。 [warning] **Trivia**: String Interpolation before Haxe 3 String Interpolation has been a Haxe feature since version 2.09. Back then, the macro Std.format had to be used, being both slower and less comfortable than the new string interpolation syntax.

6.6.数组推导

Haxe中的数组推导使用现有的语法使数组可以更简洁的初始化。它通过 for 或者 while循环构造：

> Array comprehension in Haxe uses existing syntax to allow concise initialization of arrays. It is identified by for or while constructs:

```
class Main {
    static public function main() {
        var a = [for (i in 0...10) i];
        trace(a); // [0,1,2,3,4,5,6,7,8,9]

        var i = 0;
        var b = [while(i < 10) i++];
        trace(b); // [0,1,2,3,4,5,6,7,8,9]
    }
}
```

变量 a 是被初始化为一个数组，保存从0到9的数值。编译器生成的代码，添加每次循环迭代的值到数组中，所以跟下面的代码是等价的：

> Variable a is initialized to an array holding the numbers 0 to 9. The compiler generates code which adds the value of each loop iteration to the array, so the following code would be equivalent:

```
var a = [];
for (i in 0...10) a.push(i);
```

变量 b 是初始化为一个数组，保存同样的值，但是通过一个不同的推导样式，使用了 while 循环而不是 for 。再一次，跟如下的代码是等效的：

> Variable b is initialized toan array with the same values, but through a different comprehension style using while instead of for. Again, the following code would be equivalent:

```
var i = 0;
var a = [];
while(i < 10) a.push(i++);
```

循环表达式可以是任何类型，包括条件和嵌套的循环，所以如下的内容会如预期运行：

> The loop expression can be anything,including conditions and nested loops,so the following works as expected:

```
class Main {
    static public function main() {
        var a = [
            for (a in 1...11)
                for(b in 2...4)
                    if (a % b == 0)
                        a+ "/" +b
        ];
        // [2/2,3/3,4/2,6/2,6/3,8/2,9/3,10/2]
        trace(a);
    }
}
```

6.7.迭代器

使用Haxe,很容易定义定制的迭代器和可迭代数据类型。这些概念分别由类型 Iterator 和 Iterable为代表：

> With Haxe it is very easy to define custom iterators and iterable data types. These concepts are represented by the types Iterator and Iterable respectively:

```
typedef Iterator<T> = {
    function hasNext() : Bool;
    function next() : T;
}

typedef Iterable<T> = {
    function iterator() : Iterator<T>;
}
```

任何在结构上和这些类型合一（第3.5.2节）的类都可以被通过使用 for循环迭代。也就是说，如果类定义了方法 hasNet 和 next包括匹配的返回类型，都可以被认为是一个迭代器，如果它定义了一个方法 iterator 返回一个 Iterator，则被认为是可迭代类型。

> Any class (2.3) which structurally unifies (3.5.2) with one of these types can be iterated over usingafor-loop(5.13). Thatis,if the class defines methods hasNext and next with matching return types it is considered an iterator,if it defines a method iterator returning an Iterator it is considered an iterable type.

```
class MyStringIterator {
    var s:String;
    var i:Int;

    public function new(s:String) {
        this.s = s;
        i = 0;
    }

    public function hasNext() {
        return i < s.length;
    }

    public function next() {
        return s.charAt(i++);
    }
}

class Main {
    static public function main() {
        var myIt = new MyStringIterator("string");
        for (chr in myIt) {
            trace(chr);
        }
    }
}
```

本例中的类型 MyStringIterator 有资格作为迭代器：它定义了一个方法 hasNext，返回Bool，和一个方法 next，返回String，使它兼容于 Iterator。main方法实例化了它，然后进行迭代。

> The type MyStringIterator in this example qualifies as iterator: It defines a method hasNext returning Bool and a method next returning String, making it compatible with Iterator. The main method instantiates it, then iterates over it.

```
class MyArrayWrap<T> {
    var a:Array<T>;
    public function new(a:Array<T>) {
        this.a = a;
    }

    public function iterator() {
        return a.iterator();
    }
}

class Main {
    static public function main() {
        var myWrap = new MyArrayWrap([1, 2, 3]);
        for (elt in myWrap) {
            trace(elt);
        }
    }
}
```

这里我们没有设置一个像上利中完整的迭代器，而是定义MyArrayWrap 有一个方法 iterator ，有效的转发包装的Array类型的iterator方法。

> Here we do not setup a full iterator like in the previous example, but instead define that the MyArrayWrap has a method iterator, effectively forwarding the iterator method of the wrapped Array type.

6.8.函数绑定

Haxe 3 允许绑定函数通过部分参数应用。每个函数类型可以被认为是有一个bind字段，可以被调用，传递需要数量的参数来创建一个新的函数。下面展示：

> Haxe 3 allows binding functions with partially applied arguments. Each function type can be considered to have a bind field, which can be called with the desired number of arguments in order to create a new function. This is demonstrated here:

```
class Main {
    static public function main() {
        var map = new haxe.ds.IntMap<String>();
        var f = map.set.bind(_, "12");
        $type(map.set); // Int -> String -> Void
        $type(f); // Int -> Void
        f(1);
        f(2);
        f(3);
        trace(map); // {1 => 12, 2 => 12, 3 => 12}
    }
}
```

第4行绑定函数 map.set 到一个变量 f ，然后应用 12 作为第二个参数。下划线用来表示这个参数没有绑定，通过对比map.set 和f类型展示：绑定的字符串参数被从类型中有效的切断，转换一个 Int->String->Void 类型为 Int->Void 。

> Line 4 binds the function map.set to a variable named f, and applies 12 as second argument. The underscore _ is used to denote that this argument is not bound, which is shown by comparing the types of map.set and f: The bound String argument is effectively cut from the type, turning a Int->String->Void type into Int->Void.

调用 f(1) 然后实际上调用了 map.set(1,"12")，调用 f(2) 和 f(3) 也是类似的。最后一行证明所有三个索引真实的映射到值 “12”。

> A call to f(1) then actually invokes map.set(1, "12"), the calls to f(2) and f(3) are analogous. The last line proves that all three indices indeed are mapped to the value "12".

下划线 _ 可以被跳过对于后续的参数，所以第一个参数可以被使用 map.set.bind(1)绑定，生成一个 String->Void 函数，在调用中设置一个新的索引1的值。

> The underscore _ can be skipped for trailing arguments,so the first argument could be bound through map.set.bind(1),yielding a String->Void function that sets a new value for index 1 on invocation.

回调

> 花絮：回调 Haxe 3以前，曾经使用一个 callback关键字，可以被调用为一个函数参数然后后面跟任何数量的绑定参数。这个名字源于一个常见的用法，是一个回调函数被创建和 this 对象被绑定。回调只允许绑定为参数从左到右，因为不支持下划线。使用下划线的选择有一些争议和一些其他的建议，但是没有被认为是很优越的。毕竟，下划线至少看起来像是说“在这里填上值”，很好的描述了它的语义。 [warning] **Trivia**: Callback Prior to Haxe 3, Haxe used to know a callback-keyword which could be called with a function argument followed by any number of binding arguments. The name originated from a common usage were a callback-function is created with the this-object being bound. Callback would allow binding of arguments

> only from left to right as there was no support for the underscore . *The choice to use an underscore was controversial and several other suggestions were made, none of which were considered superior. After all, the underscore* at least looks like it's saying "fill value in here", which nicely describes its semantics.

6.9.元数据

一些构造可以使用定制的元数据属性化：

> Several constructs can be attributed with custom metadata:

- 类和枚举的声明
- 类字段
- 枚举构造函数
- 表达式

> - class and enum declarations
> - Class fields
> - Enum constructors
> - Expressions

这些元数据信息可以被在运行时获得，通过 haxe.rtti.Meta API：

> These metadata information can be obtained at runtime through the haxe.rtti.Meta API:

```
import haxe.rtti.Meta;

@author("Nicolas")
@debug
class MyClass {
    @range(1, 8)
    var value:Int;

    @broken
    @:noCompletion
    static function method() { }
}

class Main {
    static public function main() {
        // { author : ["Nicolas"], debug : null }
        trace(Meta.getType(MyClass));
        // [1,8]
        trace(Meta.getFields(MyClass).value.range);
        // { broken: null }
        trace(Meta.getStatics(MyClass).method);
    }
}
```

我们可以简单的识别元数据通过开始的 @ 字符，后跟元数据的名称，和可选的，通过一些逗号分隔的包括在括号中的常量参数。

> We can easily identify metadata by the leading @ character, followed by the metadata name and, optionally, by a number of comma-separated constant arguments enclosed in parentheses.

- 类 MyClass 有一个 author 元数据，带有一个单独的String参数“Nicolas”，还有一个 debug 元数据，没有参数。
- 成员变量值有一个 range 元数据，为两个Int参数，1和8.
- 静态method 方法有一个broken元数据，没有参数，还有一个 :noCompletion 元数据，没有参数。

> - Class MyClass has an author metadata with a single String argument "Nicolas", as well as a debug

> metadata without arguments.
> - The member variable value has a range metadata with two Int arguments 1 and 8.
> - The static method method has a broken metadata without arguments,as well as a :noCompletion metadata without arguments.

main方法访问这些元数据的值可以使用它们的API。输出揭示了获得的数据的结构：

> The main method accesses these metadata values using their API. The output reveals the structure of the obtained data:

- 每个元数据都有个字段，字段名是元数据的名称。
- 字段值对应元数据参数。如果没有参数，字段值为 null 。否则字段值是一个数组，每个参数作为一个元素。
- 冒号 : 开头的元数据是省略的。这个类型的元数据作为编译器元数据存在。

> - There is a field for each metadata, with the field name being the metadata name.
> - The field values correspond to the metadata arguments. If there are no arguments,the field value is null. Otherwise the field value is an array with one element per argument.
> - Metadata starting with : is omitted. This kind of metadata is known as compiler metadata.

元数据参数接受的值为：

> Allowed values for metadata arguments are:

- 常量（第5.2节）
- 数组声明（第5.5节）（如果所有它们的元素具有资格）
- 对象声明（第5.6节）（如果所有它们的字段值有资格）

> - Constants (5.2)
> - Arrays declarations (5.5) (if all their elements qualify)
> - Object declarations (5.6) (if all their field values qualify)

内建编译器元数据：一个所有定义的元数据详尽的列表可以通过运行 haxe --help-metas 从命令行获取。

> **Built-in Compiler Metadata** An exhaustive list of all defined metadata can be obtained by running haxe --help-metas from command line.

也可以在元数据列表（第8.1节）查看。

> See also the Compiler Metadata list (8.1).

6.10.访问控制

当基础的可见性（第4.4.1节）选项不满足需求时可以使用访问控制。访问控制在类层面上和字段层面上都适用，其中涉及两个方向上的访问控制：

> Access control can be used if the basic visibility (4.4.1) options are not sufficient. It is applicable at class-level and at field-level and knows two directions:

允许访问：目标可通过 `:allow(target)` 元数据（6.9）提供给一个给定的类或字段进行访问。 强制访问：可在一个类或字段中通过 `:access(target)` 对给定目标进行强制访问。

> **Allowing access**: The target is granted access to the given class or field by using the :allow(target) metadata (6.9). **Forcing access**: A target is forced to allow access to the given class or field by using the :access(target) metadata (6.9).

其中的 **target** 是一个点路径（dot-path），在不同语境下它可能是：

- 一个类字段，
- 一个类或者一个抽象类型，
- 或者是一个包

> In this context, a target can be the dot-path (3.7) to
>
> - a class field,
> - a class or abstract type, or
> - a package.

target 与当前上下文的导入无关，所以必须是一个完整的点路径进行表示。

> Target does not respect imports, so the fully qualified path has to be used.

如果被访问控制所修饰的是一个类或者一个抽象类型，访问控制会被扩展至该类型的所有字段上。同样地，如果它是一个包，访问控制将会扩展至这个包内所有的类型、以及这些类型的所有字段上。

> If it is a class or abstract type, access modification extends to all fields of that type. Likewise, if it is a package, access modification extends to all types of that package and recursively to all fields of these types.

```
@:allow(Main)
class MyClass {
    static private var foo: Int;
}

class Main {
    static public function main() {
        MyClass.foo;
    }
}
```

这里， `MyClass.foo` 字段可以在 `main` 方法中被访问，因为 `MyClass` 被 `@:allow(Main)` 所修饰。你也可以将修饰改为 `@:allow(Main.main)` ，而且这两种修饰也都可以修饰于字段 `foo` 之上而不是类上：

> Here, MyClass.foo can be accessed from the main-method because MyClass is annotated with @:allow(Main). This would also work with @:allow(Main.main) and both versions could alternatively be annotated to the field foo instead of the class MyClass:

```
class MyClass {
    @:allow(Main.main)
    static private var foo: Int;
}

class Main {
    static public function main() {
        MyClass.foo;
    }
}
```

而如果一个类的设计不允许被修饰为这类访问控制，那么可以使用强制访问来进行访问：

> If a type cannot be modified to allow this kind of access, the accessing method may force access:

```
class MyClass {
    static private var foo: Int;
}

class Main {
    @:access(MyClass.foo)
    static public function main() {
        MyClass.foo;
    }
}
```

通过 `@access(MyClass.foo)` 访问控制修饰使得 `main` 方法可以越过 `foo` 字段的可见性修饰。

> The @:access(MyClass.foo) annotation effectively subverts the visibility of the foo field within the main-method.

元数据的选择

> 花絮：元数据的选择 访问控制的特性使用 Haxe 的元数据提供，而不是通过额外的语法提供。有如下几个理由： 额外的语法通常使得语言解析变得更加复杂，同时会导致加入太多关键字。 额外的语法需要另外的学习成本。 元数据语法足够灵活，且可以通过它来实现这一需要。 元数据可以通过Haxe的宏进行 访问/生成/修改。
>
> 当然，使用元数据语法的弊端是，如果元数据的关键字拼写错误（例如@:acesss）或者类/包名拼写错误，将不会得到错误报告。尽管如此，如果你尝试访问一个不被允许访问的私有字段时，编译器会报错，因为这不可能是一个不被觉察的错误。
>
> **Trivia**: On the choice of metadata The access control language feature uses the Haxe metadata syntax instead of additional language-specific syntax. There are several reasons for that:
>
> - Additional syntax often adds complexity to the language parsing, and also adds (too) many keywords.
> - Additional syntax requires additional learning by the language user,whereas metadata syntax is something that is already known.
> - The metadata syntax is flexible enough to allow extension of this feature.
> - The metadata can be accessed/generated/modified by Haxe macros. Of course, the main drawback of using metadata syntax is that you get no error report in case you misspell either the metadata key (@:acesss for instance) or the class/package name. However, with this feature you will get an error when you try to access a private field that you are not allowed to, therefore there is no possibility for silent errors.

Haxe 3.1.0 以后

> **Since Haxe 3.1.0**

如果允许访问一个 接口（第2.3.3节），它延伸到所有实现这个接口的类：

> If access is allowed to an interface(2.3.3),it extends to all classes implementing that interface:

```
class MyClass {
    @:allow(I)
    static private var foo: Int;
}

interface I { }

class Main implements I {
    static public function main() {
        MyClass.foo;
    }
}
```

对于访问授权父类，也是同样的，这种情况会延伸到所有的子类：

> This is also true for access granted to parent classes, in which case it extends to all child classes.

破坏性功能

> 花絮：破坏性功能 子类和实现类的访问扩展只支持Haxe 3.0以后。当编写这本手册时发现这部分的访问控制实现是容易缺失的。 [warning] **Trivia**: Broken feature Access extension to child classes and implementing classes was supposed to work in Haxe 3.0 and even documented accordingly. While writing this manual it was found that this part of the access control implementation was simply missing.

6.11.内联构造函数

Haxe 3.1.0 以后

> **Since Haxe 3.1.0**

如果一个构造函数声明为 内联（第4.4.2），编译器在某些情况下会尝试优化它。有下面几个需要：

> If a constructor is declared to be inline (4.4.2), the compiler may try to optimize it away in certain situations. There are several requirements for this to work:

- 构造函数调用的结果必须直接分配到一个局部变量
- 构造函数字段的表达式必须只包含给它的字段的赋值。

> - The result of the constructor call must be directly assigned to a local variable.
> - The expression of the constructor field must only contain assignments to its fields.

下面的例子演示了内联构造函数：

> The following example demonstrates constructor inlining:

```
class Point {
    public var x:Float;
    public var y:Float;

    public inline function new(x:Float, y:Float) {
        this.x = x;
        this.y = y;
    }
}

class Main {
    static public function main() {
        var pt = new Point(1.2, 9.3);
    }
}
```

看一下JavaScript的输出，揭示了效果：

> A look at the JavaScript output reveals the effect:

```
Main.main = function() {
    var pt_x = 1.2;
    var pt_y = 9.3;
};
```

7.编译器用法

基本使用 Haxe编译器通常从命令行调用，使用一些参数来回答两个问题：

> **Basic Usage** The Haxe Compiler is typically invoked from command line with several arguments which have to answer two questions:

- 什么应该被编译？
- 输出应该是什么？

> - What should be compiled?
> - What should the output be?

要回答第一个问题，通常通过 -cp path 参数提供一个类路径就足够了，主要的类通过 -main dot_path 参数被编译。Haxe编译器然后解析main类的文件，开始编译。

> To answer the first question, it is usually sufficient to provide a class path via the -cp path argument, along with the main class to be compiled via the -main dot_path argument. The Haxe Compiler then resolves the main class file and begins compilation.

第二个问题通常归结要提供一个特定的描述目标语言的参数。每个Haxe目标语言都有一个专用的命令行开关，例如 -js file_name 用于JavaScript，-php directory用于PHP。根据目标语言的性质，参数值可能是一个文件名（对于-js，-swf和neko），也可能是一个目录路径。

> The second question usually comes down to providing an argument specifying the desired target. Each Haxe target has a dedicated command line switch, such as -js file_name for JavaScript and -php directory for PHP. Depending on the nature of the target, the argument value is either a file name (for -js, -swf and neko) or a directory path.

一般的参数 输入：

> Common arguments *Input*:

- -cp path ：添加一个类路径，可能是可以被发现的一个 .hx 源文件或者一个包（子目录）。
- -lib library_name ：添加一个 Haxelib（第11章）库。默认，本地Haxelib仓库中最新的版本被使用。要使用特定版本，可以使用 -lib library_name:version 。
- -main dot_path ：设置主类。

> **-cp path** Addsaclasspathwhere .hx source files or packages (sub-directories) can be found. **-lib library_name** Adds a Haxelib (Chapter 11) library. By default the most recent version in the local Haxelib repository is used. To use specific version, -lib library_name:version can be used. **-main dot_path** Sets the main class.

输出:

> *output*:

-js filename：生成JavaScript（第12.1节）源代码到指定文件。 -as3 directory：生成ActionScript 3源代码到指定目录。 -swf file_name：生成指定文件为Flash（第12.2节） .swf 文件。 -neko file_name : 生成Neko（第12.3节）二进制到指定文件。 -php directory：生成PHP（第12.4节）源代码到指定目录。 -cpp directory：生成 C++（第12.5节）源代码到指定目录，并使用原生C++编译器编译。 -cs directory：生成C#（第12.7节）源代码到指定目录。 -java directory：生成Java（第12.6节）源代码到指定目录，并使用Java编译器编译。 -python file_name：生成Python（第12.8节）源代码到指定文件。

> **-js file_name** Generates JavaScript (12.1) source code in specified file. **-as3 directory** Generates ActionScript 3 source code in specified directory. **-swf file_name** Generates the specified file as Flash (12.2) .swf. **-neko file_name** Generates Neko (12.3) binary as specified file. **-php directory** Generates PHP (12.4) source code in specified directory. **-cpp directory** Generates C++(12.5) source code in specified directory and compiles it using native C++ compiler. **-cs directory** Generates C# (12.7) source code in specified directory. **-java directory** Generates Java (12.6) source code in specified directory and compiles it using the Java Compiler. **-python file_name** Generates Python (12.8) source code in the specified file.

7.1HXML

Compiler arguments can be stored in a .hxml file and can be executed with `haxe`. In hxml it is possible to use newlines and comments which makes it easier to maintain Haxe build configurations. It is possible to supply more arguments after the hxml file, e.g. `haxe build.hxml --debug`.

Example:

This example has a configuration which compiles the class file `website.HomePage.hx` to JavaScript into a file called `bin/homepage.js`, which is located in the `src` class path. And uses full dead code elimination.

```
--class-path src
--dce full
--js bin/homepage.js
--main website.HomePage
```

Multiple build compilations

Hxml configurations allow multiple compilation using these arguments:

- `--next` Separate several Haxe compilations.
- `--each` Append preceding parameters to all haxe compilations separated by `--next`. This reduces the repeating params.

Example:

This example has a configuration which compiles three different classes into their own JavaScript files. Each build uses `src` as class path and uses full dead code elimination.

```
--class-path src
--dce full

--each

--js bin/homepage.js
--main website.HomePage

--next

--js bin/gallery.js
--main website.GalleryPage

--next

--js bin/contact.js
--main website.ContactPage
```

Comments inside hxml

Inside .hxml files use a hash (i.e. `#`) to comment out the rest of the line.

Calling build configurations inside HXML:

It is possible to create a configuration that looks like this:

```
build-server.hxml
```

```
--next
build-website.hxml
--next
build-game.hxml
```

> Comment: There is a case when library's directory's hxml files can override your project's hxml during compilation. To avoid this include your `-lib library` before `-cp src -main Main`. See this haxe#7683 issue for more details.

7.2.编译器标记

从Haxe 3.0开始，你可以通过运行 haxe --help-defines 获得支持的编译器标记（第6.1节）列表。

> Starting from Haxe 3.0, you can get the list of supported compiler flags (6.1) by running `haxe --help-defines`

标记	描述				
absolute-path	打印绝对文件路径到trace 输出				
advanced-telemetry	允许SWF被Monocle工具实测				
analyzer	使用静态的分析仪用于优化（实验）				
as3	当输出flash9 as3源代码时定义				
check-xml-proxy	检查xml代理用到的字段				
core-api	定义到核心api上下文				
core-api-serialize	使用C#中的Serializable 属性标记一些生成的代码 api 类				
cppia	生成实验性的c++指令集				
dce=[mode:std\	full\	no](mode:std\	full\	no)（竖线因编辑器问题显式不正常）	设置无用代码消除（第8.2节）模式（默认为std）
dce-debug	显式无用代码消除（第8.2节）日志				
debug	当使用 -debug编译时激活				
display	编译过程中激活				
dll-export	GenCPP 实验性链接				
dll-import	GenCPP 实验性链接				
doc-gen	不执行任何 移除/改变， 以正确的生成文档				
dump	为内部的调试转储完全类型化的 AST到一个转储子目录 - 使用dump-pretty进行Haxe风格的格式化。				
dump-dependencies	转储类依赖关系到一个转储目录				
dump-ignore-var-ids	从不美观的dumps移除变量的IDs（有助于diff）				
fdb	为FDB交互调试启用完整的flash调试信息				

file-extension	输出c++源代码文件名后缀
flash-strict	flash目标的严格类型
flash-use-stage	保留SWF库的初始阶段
force-lib-check	强制编译器检查 -net-lib 和 -java-lib 添加的类（内部）
force-native-property	Tag all properties with :nativeProperty metadata for 3.1 compatibility
format-warning	Print a warning for each formated string, for 2.x compatibility
gencommon-debug	GenCommon internal
haxe-boot	Given the name 'haxe' to the flash boot class instead of a generated name
haxe-ver	当前Haxe版本值
hxcpp-api-level	Provided to allow compatibility between hxcpp versions
include-prefix	prepend path to generated include files
interp	The code is compiled to be run with --interp
java-ver=[version:5-7]	设置目标的Java版本
js-classic	Don't use a function wrapper and strict mode in JS output
js-es5	Generate JS for ES5-compliant runtimes
js-unflatten	Generate nested objects for packages and types
keep-old-output	Keep old source files in the output directory (for C#/Java)
loop-unroll-max-cost	Maximum cost (number of expressions * iterations) before loop unrolling is canceled (default 250)
macro	Defined when code is compiled in the macro context
macro-times	Display per-macro timing when used with --times
net-ver=version:20-45	设置目标的.NET版本
net-target=	Sets the .NET target. Defaults to net. xbox, micro (Micro Framework, compact (Compact Framework) are some valid values
neko-source	Output neko source instead of bytecode
neko-v1	Keep Neko 1.x compatibility

network-sandbox	Use local network sandbox instead of local file access one
no-compilation	Disable CPP final compilation
no-copt	Disable completion optimization (for debug purposes)
no-debug	Remove all debug macros from cpp output
no-deprecation-warnings	Do not warn if fields annotated with @:deprecated are used
no-flash-override	Change overrides on some basic classes into HX suffixed methods flash only
no-opt	Disable optimizations
no-pattern-matching	Disable pattern matching
no-inline	Disable inlining
no-root	GenCS internal
no-macro-cache	Disable macro context caching
no-simplify	Disable simplification filter
no-swf-compress	Disable SWF output compression
no-traces	Disable all trace calls
php-prefix	Compiled with --php-prefix
real-position	Disables haxe source mapping when targetting C#
replace-files	GenCommon internal
scriptable	GenCPP internal
shallow-expose	Expose types to surrounding scope of Haxe generated closure without writing to window object
source-map-content	Include the hx sources as part of the JS source map
swc	Output a SWC instead of a SWF
swf-compress-level=level:1-9	Set the amount of compression for the SWF output
swf-debug-password=	Set a password for debugging. The password field is encrypted by using the MD5 algorithm and prevents unauthorised debugging of your swf. Without this flag -D fdb will use no password.
swf-direct-blit	Use hardware acceleration to blit graphics
swf-gpu	Use GPU compositing features when drawing graphics

swf-metadata=	Include contents of as metadata in the swf.
swf-preloader-frame	Insert empty first frame in swf. To be used together with -D flash-use-stage and -swf-lib
swf-protected	Compile Haxe private as protected in the SWF instead of public
swf-script-timeout	Maximum ActionScript processing time before script stuck dialog box displays (in seconds)
swf-use-doabc	Use DoAbc swf-tag instead of DoAbcDefine
sys	Defined for all system platforms
unsafe	Allow unsafe code when targeting C#
use-nekoc	Use nekoc compiler instead of internal one
use-rtti-doc	Allows access to documentation during compilation
vcproj	GenCPP internal

8.编译器功能

本章内容：

8.1.内建编译器元数据

从Haxe 3.0开始，你可以运行 --help-metas 来获得定义的编译器元数据的列表。

全局元数据：

元数据	描述	平台
@:abi	Function ABI/calling convention	cpp
@:abstract	Sets the underlying class implementation as abstract type	cs java
@:access (Target path)	Forces private access to package type or field, see Access Control	all
@:allow (Target path)	Allows private access from package type or field, see Access Control	all
@:analyzer	Used to configure the static analyzer	all
@:annotation	Annotation (@interface) definitions on -java-lib imports will be annotated with this metadata. Has no effect on types compiled by Haxe	java
@:arrayAccess	Allows Array access on an abstract	all
@:autoBuild (Build macro call)	Extends @:build metadata to all extending and implementing classes. See Macro autobuild	all
@:bind	Override Swf class declaration	flash
@:bitmap (Bitmap file path)	_Embeds given bitmap data into the class (must extend flash.display.BitmapData)	flash
@:bridgeProperties	Creates native property bridges for all Haxe properties in this class	cs
@:build (Build macro call)	Builds a class or enum from a macro. See Type Building	all
@:buildXml	Specify xml data to be injected into Build.xml	cpp
@:callable	Abstract forwards call to its underlying type	all
@:classCode	Used to inject platform-native code into a class	cs java
@:commutative	Declares an abstract operator as commutative	all
@:compilerGenerated	Marks a field as generated by the compiler. Shouldn't be used by the end user	cs java
@:coreApi	Identifies this class as a core api class (forces Api check)	all
@:coreType	Identifies an abstract as core type so that it requires no implementation	all
@:cppFileCode	Code to be injected into generated cpp file	cpp
@:cppInclude	File to be included in generated cpp file	cpp
@:cppNamespaceCode		cpp
	Forces Dead Code Elimination even when not -dce full is	

@:dce	specified	all
@:debug	Forces debug information to be generated into the Swf even without -debug	flash
@:decl		cpp
@:defParam		all
@:delegate	Automatically added by -net-lib on delegates	cs
@:depend		cpp
@:deprecated	Automatically added by -java-lib on class fields annotated with @Deprecated annotation. Has no effect on types compiled by Haxe	java
@:event	Automatically added by -net-lib on events. Has no effect on types compiled by Haxe	cs
@:enum	Defines finite value sets to abstract definitions. See enum abstracts	all
@:expose (?Name=Class path)	Makes the class available on the window object or exports for node.js. See exposing Haxe classes for JavaScript	js
@:extern	Marks the field as extern so it is not generated	all
@:fakeEnum (Type name)	Treat enum as collection of values of the specified type	all
@:file(File path)	Includes a given binary file into the target Swf and associates it with the class (must extend flash.utils.ByteArray)	flash
@:final	Prevents a class from being extended	all
@:font (TTF path Range String)	Embeds the given TrueType font into the class (must extend flash.text.Font)	flash
@:forward (List of field names)	Forwards field access to underlying type	all
@:from	Specifies that the field of the abstract is a cast operation from the type identified in the function. See Implicit Casts	all
@:functionCode		cpp
@:functionTailCode		cpp
@:generic	Marks a class or class field as generic so each type parameter combination generates its own type/field	all
@:genericBuild	Builds instances of a type using the specified macro	all
@:getter (Class field name)	Generates a native getter function on the given field	flash
@:hack	Allows extending classes marked as @:final	all
@:headerClassCode	Code to be injected into the generated class, in the header	cpp
@:headerCode	Code to be injected into the generated header file	cpp
@:headerNamespaceCode		cpp
@:hxGen	Annotates that an extern class was generated by Haxe	cs java
@:ifFeature (Feature name)	Causes a field to be kept by DCE if the given feature is part of the compilation	all

@:include		cpp
@:initPackage		all
@:internal	Generates the annotated field/class with internal access	cs java
@:isVar	Forces a physical field to be generated for properties that otherwise would not require one	all
@:javaCanonical (Output type package,Output type name)	Used by the Java target to annotate the canonical path of the type	java
@:jsRequire	Generate javascript module require expression for given extern	js
@:keep	Causes a field or type to be kept by DCE	all
@:keepInit	Causes a class to be kept by DCE even if all its field are removed	all
@:keepSub	Extends @:keep metadata to all implementing and extending classes	all
@:macro	(deprecated)	all
@:mergeBlock	Merge the annotated block into the current scope	all
@:meta	Internally used to mark a class field as being the metadata field	all
@:multiType (Relevant type parameters)	Specifies that an abstract chooses its this-type from its @:to functions	all
@:native (Output type path)	Rewrites the path of a class or enum during generation	all
@:nativeChildren	Annotates that all children from a type should be treated as if it were an extern definition - platform native	cs java
@:nativeGen	Annotates that a type should be treated as if it were an extern definition - platform native	cs java
@:nativeProperty	Use native properties which will execute even with dynamic usage	cpp
@:noCompletion	Prevents the compiler from suggesting completion on this field	all
@:noDebug	Does not generate debug information into the Swf even if -debug is set	flash
@:noDoc	Prevents a type from being included in documentation generation	all
@:noImportGlobal	Prevents a static field from being imported with import Class.*	all
@:noPrivateAccess	Disallow private access to anything for the annotated expression	all
@:noStack		cpp
@:noUsing	Prevents a field from being used with using	all
@:nonVirtual	Declares function to be non-virtual	cpp
@:notNull	Declares an abstract type as not accepting null values	all

@:ns	Internally used by the Swf generator to handle namespaces	flash
@:op (The operation)	Declares an abstract field as being an operator overload	all
@:optional	Marks the field of a structure as optional. See Optional Arguments	all
@:overload (Function specification)	Allows the field to be called with different argument types. Function specification cannot be an expression	all
@:privateAccess	Allow private access to anything for the annotated expression	all
@:property	Marks a property field to be compiled as a native C# property	cs
@:protected	Marks a class field as being protected	all
@:public	Marks a class field as being public	all
@:publicFields	Forces all class fields of inheriting classes to be public	all
@:pythonImport	Generates python import statement for extern classes	python
@:readOnly	Generates a field with the readonly native keyword	cs
@:remove	Causes an interface to be removed from all implementing classes before generation	all
@:require (Compiler flag to check)	Allows access to a field only if the specified compiler flag is set	all
@:rtti	Adds runtime type informations. See RTTI	all
@:runtime		all
@:runtimeValue	Marks an abstract as being a runtime value	all
@:selfCall	Translates method calls into calling object directly	js
@:setter (Class field name)	Generates a native setter function on the given field	flash
@:sound (File path)	Includes a given .wav or .mp3 file into the target Swf and associates it with the class (must extend flash.media.Sound)	flash
@:sourceFile	Source code filename for external class	cpp
@:strict	Used to declare a native C# attribute or a native Java metadata. Is type checked	cs java
@:struct	Marks a class definition as a struct	cs
@:structAccess	Marks an extern class as using struct access('.') not pointer('->')	cpp
@:suppressWarnings	Adds a SuppressWarnings annotation for the generated Java class	java
@:throws (Type as String)	Adds a throws declaration to the generated function	java
@:to	Specifies that the field of the abstract is a cast operation to the type identified in the function. See Implicit Casts	all
@:transient	Adds the transient flag to the class field	java
@:unbound	Compiler internal to denote unbounded global variable	all
@:unifyMinDynamic	Allows a collection of types to unify to Dynamic	all
@:unreflective		cpp

@:unreflective		cpp
@:unsafe	Declares a class or a method with the C#'s unsafe flag	cs
@:usage		all
@:value	Used to store default values for fields and function arguments	all
@:void	Use Cpp native 'void' return type	cpp
@:volatile		cs java

8.2.无用代码消除

无用代码消除或者DCE是一个编译器功能，它从输出中删除未使用的代码。在类型检查之后，编译器执行DCE入口点（通常是main方法）并递归的确定哪些字段和类型被使用了。使用的字段相应进行标记，未标记的字段然后从它们的类中移除。

DCE有三个模式，当调用命令行的时候设置：

`-dce std` ：只有Haxe标准库中的类被DCE影响。这是所有目标语言默认的设置。 `-dce no` ：不执行任何DCE。
`-dce full` ：所有的类都被DCE影响。

DCE算法和类型化算法运作良好，但是当 dynamic（第2.7节）或者 反射（第10.7节）被使用时可能失败。这可能需要通过如下元数据归类明确标记的字段或者类被使用：

- @:keep ： 如果使用在类上，类和所有字段一起都不会受到DCE的影响。如果用在字段上，那个字段不受DCE影响。
- @:keepSub ：如果用在一个类上，它就像 @:keep 用在一个注释的类上，以及所有子类。
- @:keepInit ：通常，一个被DCE移除所有字段的类（或者一个开始为空的）会从输出中移除。通过使用这个元数据，空类被保留。

如果一个类需要被从命令行标记 @:keep 而不是编辑它的源代码，有一个编译器宏可用： --macro keep("type dot path") 。查看 haxe.macro.Compiler.keep API 了解这个宏的详情。它会标记包、模块或者子类型被DCE保留，包含它们进行编译。 编译器根据活动模式自动使用值“std”，“no”或者“full”定义dce。这可以被用在条件编译（第6.1节）。

> 花絮：DCE重写 DCE最初在Haxe 2.07中实现。这个实现考虑过一个函数，被用在显式的类型化时。但是问题是一些功能，多数重要的接口，会使所有的类字段被类型化以验证类型安全。这有效的完全颠覆了DCE，激励了Haxe2.10中的重写。
>
> 花絮：DCE和try.haxe.org 对于JavaScript目标语言的DCE，在网站 http://try.haxe.org 被发布时可以看到巨大的改进。生成的JavaScript代码最初的感受是混合，使得更细致的选择排除哪些代码。

8.3.编译器服务

本节内容： 8.3.1 概述 8.3.2 字段访问完成 8.3.3 调用参数完成 8.3.4 类型路径完成 8.3.5 使用完成 8.3.6 定位完成 8.3.7 顶级完成 8.3.8 完成服务

8.3.1.概述

Haxe编译器丰富的类型系统（第3章）使得IDEs和编辑器难以提供精确的完成信息。在类型推断（第3.6节）和宏（第9章）之间，需要大量的工作来反复需要的处理。这就是为什么Haxe编译器带有一个内置的完成模式供第三方软件使用。

所有完成使用 `--display file@position[@mode]` 编译器参数触发。需要的参数为：

- file：要检查完成的文件。必须是一个 .hx 文件的绝对或相对路径。它不遵守任何类路径或库。
- position：指定文件中要检查完成的字节位置（不是字符位置）。
- mode：使用的完成模式（查看下面的介绍）。

我们研究如下的完成模式细节：

- 字段访问（第8.3.2节）：提供一个可以被在一个指定类型上访问的字段列表。
- 调用参数（第8.3.3节）：报告当前被调用的函数的类型。
- 类型路径（第8.3.4节）：列出子包、子类型和静态字段。
- 使用（第8.3.5节）：列出所有编译的文件中所有指定类型、字段或者变量的出现。（模式：usage）
- 位置（第8.3.6节）：报告指定类型、字段或变量被定义的位置。（模式：position）
- 顶级（第8.3.7节）：列出所有在指定位置有效的标识符。（模式：toplevel）

由于Haxe是一个非常快的编译器，依赖普通编译器的完成调用通常就足以胜任。对于较大的项目，Haxe提供一个确保只有那些实际上发生了变化，或者它们的依赖关系有任何变化的文件才会被重新编译的服务模式（第8.3.8节）。

接口上的一般注意事项

- 如果被提到的文件在感兴趣的位置包含一个 | 字符，position参数可以被设置为0 。这对于演示和测试是非常有用的，因为它允许我们忽略一个真实的IDE不得不去做的字节计数处理。本节的例子使用了这个特性。注意，这只在 | 所在的地方不是有效的语句时可用，例如，在一个点号之后（.|）和开口的括号（ (| ）之后。
- 输出是经过HTML转义的，所以 &，< 和 > 符号分别变成 `&`， `&lt;` 和 `&gt;` 。
- 否则任何文档输出是被加工的，意味着长文档可能包括包括换行和制表符，因为它们确实在源文件中。
- 当在完成模式运行，编译器不现实错误，而是尝试忽略它们或者从错误之中恢复。如果一个致命错误发生而被完成，Haxe 编译器打印错误消息而不是完成的输出。任何非XML的输出都可以被看作是一个致命错误的消息。

8.3.2.字段访问完成

字段完成在一个点号 . 字符之后触发，来列出指定类型可用的字段。编译器解析和类型化一切到完成的点，然后输出相关的信息到 标准错误输出：

```
class Main {
    public static function main() {
        trace("Hello".|
```

```
    }
}
```

如果这个文件保存为 Main.hx ， 完成可以使用命令 haxe --display Main.hx@0 来调用。输出看起来类似于这个（为简便和提高格式的可读性，我们省略了一些字段）：

```
<list>
    <i n="length">
        <t>Int</t>
        <d>
            The number of characters in ‘this‘ String.
        </d>
    </i>
    <i n="charAt">
        <t>index : Int -&gt; String</t>
        <d>
            Returns the character at position ‘index‘ of ‘this‘ String.
            If ‘index‘ is negative or exceeds ‘this.length‘, the empty String
            "" is returned.
        </d>
    </i>
    <i n="charCodeAt">
        <t>index : Int -&gt; Null&lt;Int&gt;</t>
        <d>
            Returns the character code at position ‘index‘ of ‘this‘ String.
            If ‘index‘ is negative or exceeds ‘this.length‘, null is returned.
            To obtain the character code of a single character, "x".code can
            be used instead to inline the character code at compile time.
            Note that this only works on String literals of length 1.
        </d>
    </i>
</list>
```

XML结构如下：

- 文档借点列表包含几个节点 i，每个代表一个字段
- n 属性包含字段的名称
- t 借点包含字段的类型
- d 借点包含字段的文档

Haxe 3.2.0 以后： 当使用 -D display-details编译时，每个字段另外有一个 k 属性，它可以是变量或者方法。这得以区别方法字段和变量字段，方法字段有一个函数类型。

8.3.3.调用参数完成

调用参数完成在一个开口的括号字符 (之后触发，来显示当前被调用的函数的类型。它可以用于任何函数调用以及构造函数调用：

```
class Main {
    public static function main() {
        trace("Hello".split(|
    }
}
```

如果这个文件保存为 Main.hx ， 完成通过使用命令 haxe --display Main.hx@0 来调用。输出看起来是这样：

```
<type>
    delimiter : String -&gt; Array&lt;String&gt;
</type>
```

IDEs 可以解析这个来建议，调用的函数需要一个名为 delimiter的String类型参数，返回一个 Array。

> 花絮：输出结构的问题 我们承认当前的格式需要一些烦人的手动解析。将来我们可能考虑提供一个更结构化的输出，尤其对于函数。

8.3.4.类型路径完成

类型路径完成可以出现在 import 声明（第3.7.2节）、using 声明（第6.3节）中或者任何一个类型被引用的位置。我们可以鉴别三种不同的情况：

- 包完成：这列出haxe包的所有子包以及包中的所有模块：

```
import haxe.|
<list>
    <i n="CallStack"><t></t><d></d></i>
    <i n="Constraints"><t></t><d></d></i>
    <i n="DynamicAccess"><t></t><d></d></i>
    <i n="EnumFlags"><t></t><d></d></i>
    <i n="EnumTools"><t></t><d></d></i>
    <i n="Http"><t></t><d></d></i>
    <i n="Int32"><t></t><d></d></i>
    <i n="Int64"><t></t><d></d></i>
    <i n="Json"><t></t><d></d></i>
    <i n="Log"><t></t><d></d></i>
    <i n="PosInfos"><t></t><d></d></i>
    <i n="Resource"><t></t><d></d></i>
    <i n="Serializer"><t></t><d></d></i>
    <i n="Template"><t></t><d></d></i>
    <i n="Timer"><t></t><d></d></i>
    <i n="Ucs2"><t></t><d></d></i>
    <i n="Unserializer"><t></t><d></d></i>
    <i n="Utf8"><t></t><d></d></i>
    <i n="crypto"><t></t><d></d></i>
    <i n="ds"><t></t><d></d></i>
    <i n="extern"><t></t><d></d></i>
    <i n="format"><t></t><d></d></i>
    <i n="io"><t></t><d></d></i>
    <i n="macro"><t></t><d></d></i>
    <i n="remoting"><t></t><d></d></i>
    <i n="rtti"><t></t><d></d></i>
    <i n="unit"><t></t><d></d></i>
    <i n="web"><t></t><d></d></i>
    <in="xml"><t></t><d></d></i>
    <i n="zip"><t></t><d></d></i>
</list>
```

- 导入模块完成：这列出模块 haxe.Unserializer 所有的子类型（第3.7.1节）以及所有它的公共静态字段（因为这些也可以被导入）：

```
import haxe.Unserializer.
<list>
    <i n="DEFAULT_RESOLVER">
        <t>haxe.TypeResolver</t>
        <d>
```

```
            This value can be set to use custom type resolvers.

            A type resolver finds a Class or Enum instance from a given String .
            By default, the haxe Type Api is used.

            A type resolver must provide two methods:

            1. resolveClass(name:String):Class&lt;Dynamic&gt; is called to
            determine a Class from a class name
            2. resolveEnum(name:String):Enum&lt;Dynamic&gt; is called to
            determine an Enum from an enum name

            This value is applied when a new Unserializer instance is created.
            Changing it afterwards has no effect on previously created
            instances.
        </d>
    </i>
    <i n="run">
        <t>v : String -&gt; Dynamic</t>
        <d>
            Unserializes ‘v‘ and returns the according value.

            This is a convenience function for creating a new instance of
            Unserializer with ‘v‘ as buffer and calling its unserialize()
            method once.
        </d>
    </i>
    <i n="TypeResolver"><t></t><d></d></i>
    <i n="Unserializer"><t></t><d></d></i>
</list>
```

- 其它模块完成：这列出模块haxe.Unserializer 的所有子类型（第3.7.1）：

```
using haxe.Unserializer.|
class Main {
    static public function main() {
        var x:haxe.Unserializer.|
    }
}
<list>
    <i n="TypeResolver"><t></t><d></d></i>
    <i n="Unserializer"><t></t><d></d></i>
</list>
```

8.3.5.使用完成

从**Haxe 3.2.0** 以后： 使用完成通过 “usage” 模式参数（查看 概述（第8.3.1节））开启。我们在这里使用一个局部变量演示它。注意，它可以以同样的方式用于字段和类型上：

```
class Main {
    public static function main() {
        var a = 1;
        var b = a + 1;
        trace(a);
        a.|
    }
}
```

如果这个文件保存为 Main.hx ，可以通过使用命令 haxe --display Main.hx@0@usage 来调用完成。输出是这样的：

```
<list>
    <pos>main.hx:4: characters 9-10</pos>
    <pos>main.hx:5: characters 7-8</pos>
    <pos>main.hx:6: characters 1-2</pos>
</list>
```

8.3.6.位置完成

从**Haxe 3.2.0** 以后：

位置完成通过使用 “position”模式参数（查看概述（第8.3.1节））启用。我们使用一个字段演示它的使用。注意，它可以以同样的方式用于局部变量和类型上：

```
class Main {
    static public function main() {
        "foo".split.|
}
```

如果这个文件保存为 Main.hx ，可以使用命令 haxe --display Main.hx@0@position 调用完成。输出如下：

```
<list>
    <pos>std/string.hx:124: characters 1-54</pos>
</list>
```

> 花絮：省略目标指示符的影响 在这个例子中，编译器报告标准的 实际上没有一个实现的String.hx 定义。因为我们没有指定一个目标语言，这在完成模式是允许的。如果命令行参数包括了比如 -neko neko.n ，报告位置会被 std/neko/_std/string.hx:84:lines 84-98 取代。

8.3.7.顶级完成

从**Haxe 3.2.0** 以后： 顶级完成显示Haxe编译器识别为关于指定完成位置的所有标识符。这是唯一的我们需要一个真实的位置参数来演示它的效果的完成方法：

```
class Main {
    static public function main() {
        var a = 1;
    }
}

enum MyEnum {
    MyConstructor1;
    MyConstructor2(s:String);
}
```

如果这个文件保存为 Main.hx ，可以使用命令 haxe --desplay Main.hx@63@toplevel 来调用完成。输出看起来类似于这样（为了简洁省略了一些记录）：

```
<il>
    <i k="local" t="Int">a</i>
```

```
    <i k="static" t="Void -&gt; Unknown&lt;0&gt;">main</i>
    <i k="enum" t="MyEnum">MyConstructor1</i>
    <i k="enum" t="s : String -&gt; MyEnum">MyConstructor2</i>
    <i k="package">sys</i>
    <i k="package">haxe</i>
    <i k="type" p="Int">Int</i>
    <i k="type" p="Float">Float</i>
    <i k="type" p="MyEnum">MyEnum</i>
    <i k="type" p="Main">Main</i>
</il>
```

XML结构取决于每条记录的 k 属性。在所有情况下，i 节点的值都包含了相关的名称。 局部，成员，静态，枚举，全局：t 属性保存变量或者字段的类型。 全局，类型：p 属性保存包含该类型或者字段的模块路径。

8.3.8.完成服务

要获得最佳的编译和完成速度，你可以使用 --wait 命令行参数来启动一个 Haxe 完成服务。你也可以使用 -v 来使服务打印日志。这里是一个例子：

```
haxe -v --wait 6000
```

然后你可以连接到Haxe服务器，发送命令行参数后跟一个 0 字节，再然后，读取响应（完成结果或者错误信息）。

使用 --connect 命令行参数使Haxe 发送它的完成命令到服务器，而不是直接执行它们：

```
haxe --connect 6000 myproject.hxml
```

注意，你可以在第一次发送命令行的时候使用 --cwd 参数，改变Haxe服务器的当前工作目录。通常，类路径和其它的文件跟你的项目是相关的。

它如何工作 完成服务会缓存下面的东西： 解析的文件 文件只有在它们被修改或者出现一个解析错误的时候才会再次被解析 **haxelib** 调用 前面的haxelib调用的结果可以被重用（只用于完成：它们在做编译时是被忽略的） 类型的模块 编译模块在一个成功的编译后会被缓存，并且如果它们的依赖关系没有被修改，则可以在之后的 编译/完成 中重用

你可以了解编译器花费的精确时间，和如何使用完成服务影响它们，通过添加 --times 到命令行即可。

协议 如下面的Haxe/Neko示例展示的，你可以简单的连接到服务端口，并发送所有的命令（或者每行）以0二进制字符结束。然后就可以读取结果。

宏和其它的命令可以记录不是错误的事件。从命令行中，我们可以看到 stdout 打印的和stderr打印的内容的不同。这不是sockt模式下的情况。为了区分两者，日志消息（不是错误消息）前置一个 x01 字符，消息中所有的换行符都被同样的 x01 字符取代。

警告和其它的消息也可以被认为是错误，但是是不致命的。如果一个不致命的错误发生，它会发送一个单行的 x02 消息行。

这里是一些将连接到服务并处理协议细节的代码：

```
class Test {
    static function main() {
        var newline = "\textbackslash\ n";
        var s = new neko.net.Socket();
```

```
        s.connect(new neko.net.Host("127.0.0.1"),6000);
        s.write("--cwd /my/project" + newline);
        s.write("myproject.hxml" + newline);
        s.write("\textbackslash\ 000");

        var hasError = false;
        for (line in s.read().split(newline))
        {
            switch (line.charCodeAt(0)) { 1
                case 0x01:
                    neko.Lib.print(line.substr(1).split("\ textbackslash\ x01").join(newline));
                case 0x02:
                    hasError = true;
                default:
                    neko.io.File.stderr().writeString(line + newline);
            }
        }
        if (hasError) neko.Sys.exit(1);
    }
}
```

对宏的影响 完成服务可能对宏的执行（第9章）有一些副作用。

8.4.资源

Haxe 提供 一个简单的资源嵌入系统，可以用来直接嵌入文件到编译后的应用。

然而它可能不是嵌入大型资源如图像或者音乐到应用文件中的最优选择，在嵌入小型资源如配置或者XML数据时是非常方便的。

8.4.1.嵌入资源

外部文件使用 -resource 编译器参数嵌入：

```
-resource hello_message.txt@welcome
```

@符号后面的字符串是资源标识符，用在代码中用于检索资源。如果它被省略（包括@符号一起）那么文件名则成为资源标识符。

8.4.2.检索文本资源

要检索一个嵌入的资源的内容，我们使用 haxe.Resource 的静态方法 getString ，给它传递一个资源标识符：

```
class Main {
    static function main() {
        trace(haxe.Resource.getString("welcome"));
    }
}
```

上面的代码会显示先前我们使用 welcome 作为标识符而包含的 hello_message.txt 文件的内容。

8.4.3.检索二进制资源

虽然不推荐嵌入大型二进制文件到应用中，但嵌入二进制数据仍然可能是有用的。一个嵌入的资源的二进制表现可以使用 haxe.Resource 的静态方法 getBytes 访问：

```
class Main {
    static function main() {
        var bytes = haxe.Resource.getBytes("welcome");
        trace(bytes.readString(0, bytes.length));
    }
}
```

getBytes方法返回的类型是 haxe.io.Bytes ，它是一个对象，提供对数据的个体字节的访问。

8.4.4.实现细节

如果有，Haxe使用目标平台的原生资源绑定，否则它提供自己的实现。

- Flash 资源被如ByteArray定义嵌入
- C# 资源被包含到编译后的程序集
- Java 资源被包装进结果的JAR 文件
- C++ 资源被存储到全局的字节数组常量
- JavaScript 资源被序列化为Haxe序列化格式，并存储到 haxe.Resource 类的一个静态字段
- Neko 资源被作为字符串存储到 haxe.Resource 类的一个静态字段

8.5.运行时类型信息

Haxe编译器为注解的类或者通过 :rtti 元数据注解的扩展的类生成运行时类型信息（RTTI）。这个信息被存储为一个 XML 字符串到一个字段 __rtti，并可以通过 haxe.rtti.XmlParser 进行处理。结果的结构被描述为 RTTI 结构（第8.5.1节）。

从 **Haxe 3.2.0** 以后： 类型 haxe.rtti.Rtti 已经被引入，来简化RTTI相关的工作 。检索这个信息现在非常简单：

```
@:rtti
class Main {
    var x:String;
    static function main() {
        var rtti = haxe.rtti.Rtti.getRtti(Main);
        trace(rtti);
    }
}
```

8.5.1.RTTI 结构

一般类型信息：

- path: path 类型（第3.7节）的类型路径
- module:包含这个类型的模块（第3.7节）的类型路径
- file: 包含这个类型的 .hx 文件的完整的斜线路径。在没有这个文件的时候可以为 null，例如，如果这个类型通过一个宏（第9章）来定义。
- params: 一个字符串类型的数组，表示这个类型的类型参数（第3.2节）的名称。自Haxe 3.2.0起，这不包括约束（第3.2.1节）。
- doc: 类型的文档。这个信息只有编译器标记（第6.1节） -D use_rtti_doc 被涉及的时候才可用。否则，或者如果类型没有文档，值为 null 。
- isPrivate: 是否类型为 私有（第3.7.1节）。
- platforms: 一个字符串列表，表示类型可用的目标平台。
- meta: 类型被注解的元数据。

类类型信息：

- isExtern: 是否类为 外部的（第6.2节）
- isInterface: 是否类实际上是一个接口（第2.3.3节）。
- superClass: 被它的类型路径和类型参数列表定义的它的父类。
- interfaces: 被它们的类型路径和类型参数列表定义的接口列表。
- fields: 成员类字段（第4章）列表，描述为类字段信息（第8.5.1节）。
- statics: 静态类字段列表，描述为类字段信息（第8.5.1节）。
- tdynamic: 如果没有这样的类型存在则为被通过类或者null动态实现（第2.7.2节）的类型。

枚举类型信息：

- isExtern: 枚举是否为外部的（第6.2节）。
- constructors: 枚举的构造函数列表。

抽象类型信息：

- to:一个包含隐式定义 to 转换（第2.8.1节）的数组。

- from: 一个包含隐式定义 from 转换（第2.8.1节）的数组。
- impl: 实现类的类型信息（第8.5.1节）。
- athis: 抽象类型的潜在类型（第2.8节）。

类字段信息：

- name: 字段名。
- type: 字段类型。
- isPublic: 字段是否为 public（第4.4.1节）类型。
- isOverride: 字段是否重载（第4.4.4节）另一个字段。
- doc: 字段的文档。这个信息只在如果编译器标记（第6.1节） -D use_rtti_doc 被涉及的时候可用。否则，或者如果字段没有文档，值为 null 。
- get: 字段的读取访问行为（第4.2节）。
- set: 字段的写入访问行为（第4.2节）。
- params: 一个字符串数组，表示字段拥有的类型参数（第3.2节）的名称。 从Haxe 3.2.0 起，这不包括约束（第3.2.1节）。
- platforms: 一个字符串列表表示字段可用的目标平台。
- meta: 字段被注解用的元数据。
- line: 字段定义的行号。这个信息只有字段有一个表达式的时候可用。否则值为null。
- overloads: 字段可用的重载列表，或者如果没有存在的重载则为 null 。

枚举构造函数信息： name: 构造函数的名称。 args: 构造函数的参数列表，如果没有可用参数则为null。 doc: 构造函数的文档。这个嘻嘻只有如果编译器标记（第6.1节） -D use_rtti_doc 被涉及的时候才可用。否则，或者如果构造函数没有文档，值为null。 platforms: 一个字符串列表，表示构造函数可用的目标平台。 meta: 构造函数被注解的元数据。

8.6.静态分析仪

从 **Haxe 3.3.0** 起： Haxe 3.3.0 为代码优化引入了一个新的静态分析仪。通过使用 -D analyzer 编译器标记（第7.1节）来启用，由多个模块（第8.6节）组成，可以被用编译器标记（第7.1节）进行全局配置，以及在 类型级别和字段级别被用一个编译器元数据（第8.1节）进行全局配置。

全局配置： 要全局启动一个分析仪模块，使用 -D analyzer-module 。要全局禁用一个模块，使用 -D analyzer-no-module。两种情况，“module”表示被禁用或启用的模块的名字：

```
# Global enable from command line
haxe -D analyzer-module
# Global disable from command line
haxe -D analyzer-no-module
```

本地配置： 要对一个给定的类型或者字段启动一个分析仪模块，使用 @:analyzer(module)。要禁用一个模块，使用 @:analyzer(no_module) 。两种情况，“module”表示要禁用或者启用的模块的名字：

```
@:analyzer(module)
class C {
    @:analyzer(module) function f() { } // Field-level enable
    @:analyzer(no_module) function f() { } // Field-level disable
}
@:analyzer(no_module)
class D { } // Type-level disable
```

Modules

静态分析仪当前带有如下的模块。如果 -D analyzer 被使用，除非注明否则它们是被启用的。

const_propagation : 实现稀少条件的常量传播来促进编译时知道的值到使用的地方。也侦测无效的分支。
copy_propagation : 侦测别名化其它局部变量的局部变量并相应的进行替换。

local dce : 侦测并移除无用的局部变量。

fusion : 在单一事件情况下移动变量表达式到它的使用中。在Flash和 Java目标语言中是禁用的。

purity_inference : 推断是否字段为 “纯粹的”，即，没有任何的副作用。这可以改善 fusion 模块的影响。

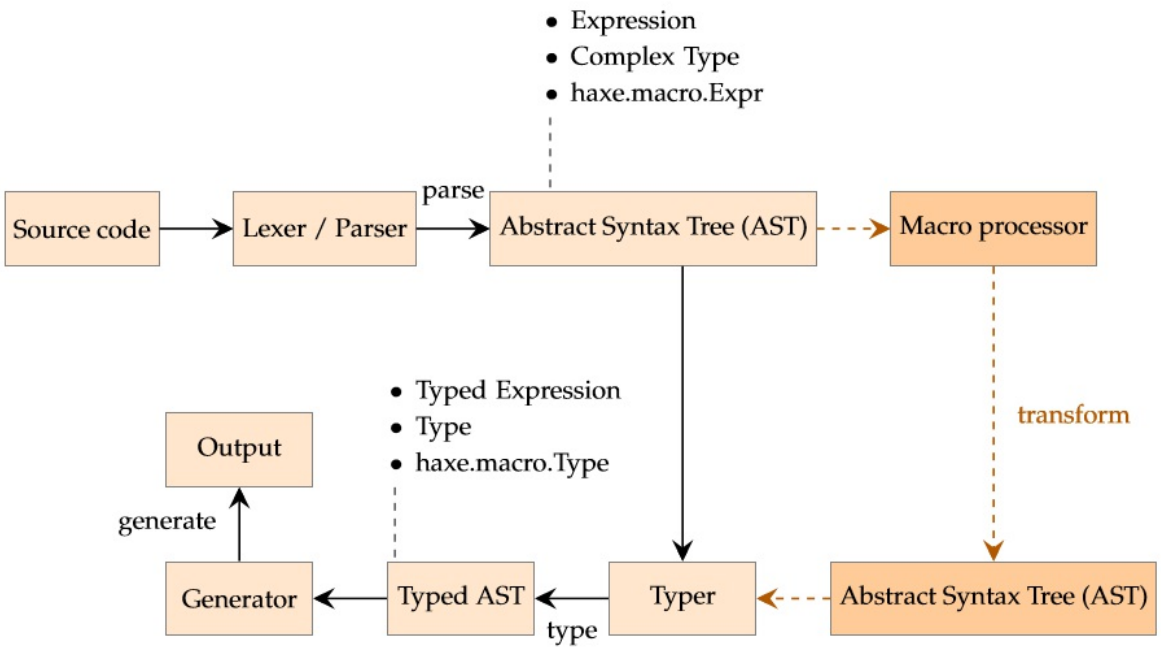
unreachable_code : 报告无法达到的代码。

9.宏

宏毫无疑问是Haxe中最先进的功能。它们经常被认为是黑魔法，只有少数人能够精通，但其实它们没有什么魔法（当然也没有黑暗）。

> 定义：抽象语法树（AST） AST是解析Haxe代码到一个类型化的结构的结果。这个结构被通过在Haxe标准库的 haxe/macro/Expr.hx 定义的类型暴露给宏。

编译中宏的规则如下：



一个基本的宏是一个语法转换。它接受0或者多个表达式（第5章）并返回一个表达式。如果一个宏被调用，它实际上从它调用的位置插入代码。在这方面，它可以被跟一个预处理，如C++中的 #define对比，但是一个Haxe 宏并不是一个文本替换工具。

我们可以识别不同种类的宏，它们运行在特定的编译阶段：

初始化宏：这些被通过命令行提供，使用 --macro 编译器参数。它们在编译器参数被处理、类型器上下文被创建之后，但在任何类型化开始之前执行（查看初始化宏（第9.7节））。

构建宏：这些是为类、枚举和抽象类型定义，通过 @:build 或者 @:autoBuild 元数据（第6.9节）定义。在每次类型化时执行，在类型被设置之后（包括跟其它类型的关系，比如类的继承），但是在它的字段被类型化之前（查看类型构建（第9.5节））。

表达式宏：这些是普通的函数，在它们被类型化之后马上执行。

9.1.宏上下文

> 定义：宏上下文 宏的上下文是宏被执行的环境。根据宏的类型，它可以被认为是一个类被构建，或者一个函数被类型化。上下文的信息可以通过 haxe.macro.Context API 获得。

Haxe 的宏根据宏的类型有对不同上下文信息的访问。除了查询这些信息，上下文也允许一些修改比如定义一个新的类型或者注册某个回调。重要的是，理解不是所有信息对所有种类的宏都可用，就如后面例子所证明的：

- 初始化宏将发现 Context.getLocal*() 方法返回 null 。没有局部类型或者方法在初始化宏的上下文中。
- 只有构建宏从 Context.getBuildFields() 方法得到正确的返回值。对于其它种类的宏没有字段被构建。
- 构建宏有一个局部类型（如果不完全的），但是没有局部方法，所以 Context.getLocalMethod() 返回 null 。

上下文API被 haxe.macro.Compiler API 补充，在初始化宏（第9.7节）中详述。而这个API可以用于所有宏的种类，必须注意初始化宏的任何外部修改。这源于未定义的构建顺序（第9.6.3节）的自然限制，可能导致如一个标记定义通过 Compiler.define() 在一个对应这个标记的条件编译检查之前或之后生效。

9.2.参数

多数时候，宏的参数是表示为一个 enum Expr的实例的表达式。这样，它们被解析但没有类型化，意味着它们可以是符合Haxe的语法规则的任何内容。宏然后可以检视它们的结构，或者使用 haxe.macro.Context.typeof() 方法（尝试）得到它们的类型。

重要的是要理解宏的参数不保证被评估，所以任何预期的副作用不保证会出现。另一方面，同样重要的是理解一个参数表达式可能被一个宏复制和多次用在返回的表达式中：

```
import haxe.macro.Expr;

class Main {
    static public function main() {
        var x = 0;
        var b = add(x++);
        trace(x); // 2
    }

    macro static function add(e:Expr) {
        return macro $e + $e;
    }
}
```

宏 add 被调用，x++作为参数，并因此使用表达式具体化（第9.3.1节）返回 x++ + x++ ，使 x 被增加两倍。

9.2.1.ExprOf

由于Expr兼容任何可能的输入，Haxe提供了一个类型 haxe.macro.ExprOf 。大多数情况下，这个类型和Expr完全相同的，但是它允许限制接受的表达式的类型。这在结合宏和静态扩展（第6.3节）时可以提供帮助：

```
import haxe.macro.Expr;
using Main;

class Main {
    static public function main() {
        identity("foo");
        identity(1);
        "foo".identity();
        // Int has no field identity
        //1.identity();
    }

    macro static function identity(e:ExprOf<String>) {
        return e;
    }
}
```

两个对 identity 的直接调用被接受，即使参数声明为 ExprOf 。这可能有点出乎意料，Int 1 被接受，但是它是关于宏参数（第9.2节）中解释的一个合乎逻辑的结论：参数表达式从不被类型化，所以它不可能让编译器使用合一（第3.5节）检查它们的兼容性。

下两行使用静态扩展（注意 using Main）的则有所不同：对于这些它是强制首先类型化左侧（“foo” 和 1）来理解identity字段访问。这使它可以检查参数类型对应的类型，使 1.identity() 不把 Main.identity() 作为一个适用的字段。

9.2.2.常数表达式

一个宏可以被声明接受常数（第5.2节）参数：

```
class Main {
    static public function main() {
        const("foo", 1, 1.5, true);
    }

    macro static function const(s:String, i:Int, f:Float, b:Bool) {
        trace(s);
        trace(i);
        trace(f);
        trace(b);
        return macro null;
    }
}
```

通过这些，就不需要绕过表达式，因为编译器可以直接使用提供的常数。

9.2.3.其它的参数

如果一个宏最后的参数是 Array类型，宏可以以数组形式接受一个任意数量的额外参数：

```
import haxe.macro.Expr;

class Main {
  static public function main() {
    myMacro("foo", a, b, c);
  }

  macro static function myMacro(e1:Expr, extra:Array<Expr>) {
    for (e in extra) {
      trace(e);
    }
    return macro null;
  }
}
```

9.3.具体化

Haxe编译器允许具体化表达式、类型和类来简化宏的使用。具体化的语法是 macro expr ， expr 是任何有效的Haxe表达式。

9.3.1.表达式具体化

表达式具体化用于以一个便捷的方式创建haxe.macro.Expr的实例。Haxe编译器接受通常的Haxe语法并翻译它到一个表达式对象。它支持几种转义机制，都是通过 $ 符号触发：

- **${}** 和 **$e{}: Expr -> Expr** 这可以被用来合并表达式。 {}界限内的表达式被执行，使用它的值进行替换。
- **$a{}: Expr -> Array** 如果用于一个 预期为Array的位置（如调用参数，块元素），$a{} 把数组作为它的值。否则它生成一个数组声明。
- **$b{}:Array->Expr** 从给定的表达式数组生成一个块级表达式。
- **$i{}:String->Expr** 从给定字符串生成一个标识符。
- **$p{}:Array->Expr** 从给定字符串数组生成一个字段表达式。
- **$v{}:Dynamic->Expr** 根据它的参数的类型生成一个表达式。这仅仅保证使用在基本类型和枚举实例。

另外，元数据 @:pos(p) 可以用来映射注解表达式的位置到 p ，而不是被具体化的位置。

这类具体化只工作于 内部结构预期一个表达式时。不允许 object.${fieldName}，但是 object.$fieldName 可以使用。这适用于所有内部构造预期一个字符串的位置：

字段访问 object.$name 变量名称 var $name = 1;

从**Haxe 3.1.0** 以后：

字段名 { $name: 1} 函数名 function $name() { } 捕获变量名 try e() catch($name:Dynamic) { }

此外，一个新的表达式可以通过提供 haxe.macro.TypePath 参数被具体化： new $typePath()

9.3.2.类型具体化

类型具体化用于以一个简便的方式创建 haxe.macro.Expr.ComplexType 的实例。通过一个 macro : Type 定义，Type 可以是任何有效的类型路径表达式。这类似于普通代码中的显式类型提示，例如对于变量，形式为 var x:Type 。

ComplexType的每个构造函数都有不同的语法：

TPath: macro : pack.Type **TFunction:** macro : Arg1 -> Arg2 -> Return **TAnonymous:** macro : { field: Type } **TParent:** macro : (Type) **TExtend:** macro : {> Type, field: Type } **TOptional:** macro : ?Type

9.3.3.类具体化

还可以使用具体化来获取一个 haxe.macro.Expr.TypeDefinition 的实例。这是通过 macro 类语法表示的，如下面显示的：

```
class Main {
```

```
  macro static function generateClass(funcName:String) {
    var c = macro class MyClass {
      public function new() { }
      public function $funcName() {
        trace($v{funcName} + " was called");
      }
    }
    haxe.macro.Context.defineType(c);
    return macro new MyClass();
  }

  public static function main() {
    var c = generateClass("myFunc");
    c.myFunc();
  }
}
```

生成的 TypeDefinition 实例通常传递到 haxe.macro.Context.defineType 来添加一个i额新的类型到调用上下文（不是宏上下文本身）。

这类的具体化可以用于获得 haxe.macro.Expr.Field的实例，可从生成的TypeDefinition的字段数组获得。

9.4.工具

Haxe标准库附带一组 工具类来简化宏的使用。这些类最好作为静态扩展使用，通过使用 haxe.macro.Tools ，可以被单独或者完整的带入到上下文中。这些类为：

- ComplexTypeTools ： 允许以一个人类可读的方式打印 ComplexType 实例 。也允许确定相对于 一个ComplexType的 Type。
- ExprTools ： 允许以一个人类可读的方式打印 Expr 实例。也允许迭代和映射表达式。
- MacroStringTools ： 提供对宏上下文中的字符串和字符串表达式有用的操作。
- TypeTools ： 允许以一个人类可读的方式打印 Type 实例。也提供一些对类型有用的操作，譬如统一（第3.5节）它们，或者得到它们的相应的 ComplexType 。

此外，haxe.macro.Printer 类有以一个人类可读格式打印各种类型的的公共方法。当调试宏的时候可以提供帮助。

> 花絮：tinkerbell 库 和 为什么使用Tools.hx 我们学习了静态扩展，使用一个模块意味着所有它的类型被带入静态扩展上下文中。事实证明，这样一个类型也可以是一个对其它类型的typedef（第3.1节）。然后编译器将这个类型当作模块的一部分，并相应的延伸静态扩展。 这种“伎俩”最初在 Juraj Kirchheim 的 tinkerbell 库用于相同的目的。Tinkerbell 在它们进入Haxe编译器和标准库很久之前就提供许多有用的宏工具。它仍然是额外的宏工具主要的库，提供其它有用的功能。

9.5.类型构建

类型构建宏和表达式宏在很多方面是不同的：

- 它们不返回表达式，而是一个数组或者类字段。它们的返回类型必须显式设置为 Array 。
- 它们的上下文环境（第9.1节）没有局部方法也没有局部变量。
- 它们的额上下文环境有构建字段，可以从 haxe.macro.Context.getBuildFields() 获得。
- 它们不会被直接调用，但是是一个类或者枚举声明上的 @:build 或者 @:autoBuild 元数据（第6.9节）的一个参数。

如下的例子演示了类型构建。注意，它被分成两个文件是有原因的：如果一个模块包含一个宏函数，它也必须类型化到 macro 上下文。 这对类型构建宏来说经常是一个问题，因为要构建的类型只能在它的不完整阶段被加载，在构建宏运行之前。我们建议总是定义类型构建宏到它们自己的模块。

```
import haxe.macro.Context;
import haxe.macro.Expr;

class TypeBuildingMacro {
  macro static public function build(fieldName:String):Array<Field> {
    var fields = Context.getBuildFields();
    var newField = {
      name: fieldName,
      doc: null,
      meta: [],
      access: [AStatic, APublic],
      kind: FVar(macro : String, macro "my default"),
      pos: Context.currentPos()
    };
    fields.push(newField);
    return fields;
  }
}
@:build(TypeBuildingMacro.build("myFunc"))
class Main {
  static public function main() {
    trace(Main.myFunc); // my default
  }
}
```

TypeBuildingMacro 的构建方法执行三步：

1. 它使用 Context.getBuildFields() 获得构建字段。
2. 它使用 funcNace 宏参数作为字段名声明一个新的 haxe.macro.expr.Field 字段。这个字段是一个 String变量默认值为 “my default”（来自于 kind 字段），而且是 public static 的（来自于 access 字段）。
3. 它添加新的字段到构建的字段数组，并返回它。

这个宏是Main 类上 @:build 元数据的参数。一旦这个类型被需要的，编译器就做以下事：

1. 它解析模块文件，包含类字段。
2. 它设置类型，包含它和其它类型通过继承和接口产生的关系。
3. 它根据 @:build 元数据 执行类型构建宏。
4. 它使用被类型构建宏返回的字段继续如常类型化类。

这允许在一个类型构建宏中随意添加和修改类字段。在我们的例子中，宏被调用，使用了一个 myFunc 参数，使 Main.myFunc 成为一个有效的字段访问。

如果一个类型构建宏不应修改任何内容，宏可以返回null 。这指示编译器没有刻意的改变，更好的是返回 Context.getBuildFields() 。

9.5.1.枚举构建

构建枚举类似于使用一个简单的映射构建类：

- 没有参数的枚举构造函数是变量字段 FVar。
- 带有参数的枚举构造函数是方法字段 FFun 。

```
import haxe.macro.Context;
import haxe.macro.Expr;

class EnumBuildingMacro {
  macro static public function build():Array<Field> {
    var noArgs = makeEnumField("A", FVar(null, null));
    var eFunc = macro function(value:Int) { };
    var fInt = switch (eFunc.expr) {
      case EFunction(_,f): f;
      case _: throw "false";
    }
    var intArg = makeEnumField("B", FFun(fInt));
    return [noArgs, intArg];
  }

  static function makeEnumField(name, kind) {
    return {
      name: name,
      doc: null,
      meta: [],
      access: [],
      kind: kind,
      pos: Context.currentPos()
    }
  }
}
@:build(EnumBuildingMacro.build())
enum E { }

class Main {
  static public function main() {
    switch(E.A) {
      case A:
      case B(v):
    }
  }
}
```

因为枚举 E 被使用一个 :build 元数据注解，调用的宏构建两个构造函数 A 和 B 到它之中。前者被添加使用的类型是 FVar(null, null)，意味着它是一个构造函数，没有参数。后者，我们使用具体化来获得 haxe.macro.Expr.Function 的一个实例，参数为一个单独的 Int 。

main 方法通过匹配它，证明了我们生成的枚举的结构。我们可以发现，生成的类型跟下面这个是等价的：

```
enum E {
  A;
  B(value:Int);
}
```

9.5.2.@:autoBuild

如果一个类有 :autoBuild 元数据，编译器生成 :build 元数据 到所有扩展类。如果一个接口有 :autoBuild 元数据，编译器生成 :build 元数据到所有实现类和所有扩展接口。注意 :autoBuild 并不是暗示 :build 到 类/接口 本身。

```
import haxe.macro.Context;
import haxe.macro.Expr;

class AutoBuildingMacro {
  macro static public
  function fromInterface():Array<Field> {
    trace("fromInterface: " + Context.getLocalType());
    return null;
  }

  macro static public
  function fromBaseClass():Array<Field> {
    trace("fromBaseClass: " + Context.getLocalType());
    return null;
  }
}
@:autoBuild(AutoBuildingMacro.fromInterface())
interface I { }

interface I2 extends I { }

@:autoBuild(AutoBuildingMacro.fromBaseClass())
class Base { }

class Main extends Base implements I2 {
  static public function main() { }
}
```

这在编译过程中输出：

```
AutoBuildingMacro.hx:6:
  fromInterface: TInst(I2,[])
AutoBuildingMacro.hx:6:
  fromInterface: TInst(Main,[])
AutoBuildingMacro.hx:11:
  fromBaseClass: TInst(Main,[])
```

重要的是记住 这些宏执行的顺序是未定义的，在构建顺序（第9.6.3节）中详述。

9.5.3.@:genericBuild

从**Haxe 3.1.0** 后： 一般构建宏每个类型化运行，已经非常强大。在一些情况下，每个类型化运行一个构建宏相反用法，即，当它实际上出现在代码中。除此之外这允许在宏中访问具体的类型参数。

@:genericBuild 就像 @:build 一样使用，通过添加它到一个类型，并使用一个宏调用作为参数：

```
import haxe.macro.Expr;
import haxe.macro.Context;
import haxe.macro.Type;

class GenericBuildMacro1 {
  static public function build() {
```

```
    switch (Context.getLocalType()) {
      case TInst(_, [t1]):
        trace(t1);
      case t:
        Context.error("Class expected", Context.currentPos());
    }
    return null;
  }
}

@:genericBuild(GenericBuildMacro1.build())
class MyType<T> { }

class Main {
  static function main() {
    var x:MyType<Int>;
    var x:MyType<String>;
  }
}
```

当运行这个例子，编译器输出 TAbstract(Int,[]) 和 TInst(String,[])，表明它确实意识到 MyType的具体类型参数。宏的逻辑可以使用这个信息来生成一个定制类型（使用 haxe.macro.Context.defineType）或者引用一个存在的类型。方便起见，我们在这里返回null，要求编译器推断这个类型。

在Haxe 3.1 中，一个 @:genericBuild 宏的返回类型必须是一个 haxe.macro.Type 。Haxe 3.2 允许（也更愿意）返回一个 haxe.macro.ComplexType ，是一个类型语法上的表示。在很多情况下这变得很容易使用，因为类型可以通过它们的路径简单的引用。

Const 类型参数 如果类型参数名称是 Const，Haxe 允许传递常量表达式（第5.2节）作为一个类型参数。这可以在 @:genericBuild 宏的上下文中被利用来从语法直接到宏传递信息：

```
import haxe.macro.Expr;
import haxe.macro.Context;
import haxe.macro.Type;

class GenericBuildMacro2 {
  static public function build() {
    switch (Context.getLocalType()) {
      case TInst(_,[TInst(_.get() => { kind: KExpr(macro $v{(s:String)}) },_)]):
        trace(s);
      case t:
        Context.error("Class expected", Context.currentPos());
    }
    return null;
  }
}
@:genericBuild(GenericBuildMacro2.build())
class MyType<Const> { }

class Main {
  static function main() {
    var x:MyType<"myfile.txt">;
  }
}
```

这里宏的逻辑可以加载一个文件并使用它的内容来生成一个定制类型。

9.6.限制

本节内容：

9.6.1Macro-in-Macro

Build macros cannot be invoked from a macro context. This means it is impossible to use a macro to create a class which will provide build macros for other classes.

since Haxe 4.0.0

It is also disallowed to invoke expression macros in a macro context.

> **Trivia: Expression macro-in-macro**
>
> Prior to Haxe 4, using expression macros inside a macro context was possible. Support was primarily dropped because such code would cause issues with the compiler cache.

9.6.2.静态扩展

静态扩展（第6.3节）和宏的概念有一些冲突：前者需要一个已知类型来确定使用的函数，而宏在类型化简单的语法之前执行。因此毫不奇怪，结合使用这两个特性可能导致问题。Haxe 3.0 会尝试转换 类型化表达式回到一个 语法表达式，并不总是可行，可能丢失一些重要信息。我们建议小心使用。

从 **Haxe 3.1.0** 以后： 静态扩展和宏的组合在 3.1.0 版本被重构。Haxe编译器甚至不设法寻找宏参数最初的表达式，而是传递一个特殊的 @:this this 表达式。而这个表达式的结构不传达信息，表达式仍然被正确的类型化：

```
import haxe.macro.Context;
import haxe.macro.Expr;

using Main;
using haxe.macro.Tools;

class Main {
  static public function main() {
    #if !macro
    var a = "foo";
    a.test();
    #end
  }

  macro static function test(e:ExprOf<String>) {
    trace(e.toString()); // @:this this
    // TInst(String,[])
    trace(Context.typeof(e));
```

```
    return e;
  }
}
```

9.6.3.构建顺序

The build order of types is unspecified and this extends to the execution order of build-macros. While certain rules can be determined, we strongly recommend to not rely on the execution order of build-macros. If type building requires multiple passes, this should be reflected directly in the macro code. In order to avoid multiple build-macro executions on the same type, the state can be stored in persistent variables or added as metadata to the type in question:

```
import haxe.macro.Context;
import haxe.macro.Expr;

#if !macro
@:autoBuild(MyMacro.build())
#end
interface I1 {}
#if !macro
@:autoBuild(MyMacro.build())
#end
interface I2 {}
class C implements I1 implements I2 {}

class MyMacro {
  macro static public function build():Array<Field> {
    var c = Context.getLocalClass().get();
    if (c.meta.has(":processed"))
      return null;
    c.meta.add(":processed", [], c.pos);
    // process here
    return null;
  }
}

class Main {
  static public function main() {}
}
```

With both interfaces `I1` and `I2` having `:autoBuild` metadata, the build macro is executed twice for class `C`. We guard against duplicate processing by adding a custom `:processed` metadata to the class, which can be checked during the second macro execution.

9.6.4.类型参数

官方文档暂无内容

9.7.初始化宏

初始化宏从命令行调用，使用 --macro callExpr(args) 命令。这注册一个回调，编译器在创建它的上下文环境后，但是在类型化 -main 的参数之前调用。之后允许以某些方式配置编译器。

如果 --macro 的参数是一个简单的标识符的调用，这个标识符在Haxe标准库中的类 haxe.macro.Compiler 中查找。它附带一些有用的初始化宏，在它的 API 中有详细介绍。

当然也可以定义自定义的初始化宏来在真正编译之前执行各种任务。这样的一个宏将通过 --macro some.Class.theMacro(args) 调用。例如，因为所有宏分享同样的上下文（第9.1节），一个初始化宏可以设置其它宏用作配置的静态字段的值。

10.标准库

本章内容：

10.1.字符串

> 类型: String 一个字符串是一个字符序列。

字符编码

使用 .code 属性到一个常量单字节，可以编译它的ASCII字节码：

```
"#".code // will compile as 35
```

查看 String API 了解更多关于它的方法的介绍。

10.1.1.字符串字面值

A string literal is a sequence of characters inside a pair of double quotes or single quotes:

```
var a = "foo";
var b = 'foo';
trace(a == b); // true
```

The only difference between the two forms is that single-quoted literals allow string interpolation.

Escape sequences

Sequence	Meaning	Unicode codepoint (decimal)	Unicode codepoint (hexadecimal)
`\t`	horizontal tab (TAB)	9	0x09
`\n`	new line (LF)	10	0x0A
`\r`	new line (CR)	13	0x0D
`\"`	double quote	34	0x22
`\'`	single quote	39	0x27
`\\\\\\\\`	backslash	92	0x5C
`\NNN`	ASCII escape where `NNN` is 3 octal digits	0 - 127	0x00 - 0x7F
`\xNN`	ASCII escape where `NN` is a pair of hexadecimal digits	0 - 127	0x00 - 0x7F
`\uNNNN`	Unicode escape where `NNNN` is 4 hexadecimal digits	0 - 65535	0x0000 - 0xFFFF
`\u{N...}`	Unicode escape where `N...` is 1-6 hexadecimal digits	0 - 1114111	0x000000 - 0x10FFFF

10.1.2.Unicode

since Haxe 4.0.0

All Haxe targets except Neko support Unicode in strings by default. The compile-time define `target.unicode` is set on targets where Unicode is supported.

A string in Haxe code represents a valid sequence of Unicode codepoints. Due to differing internal representations of strings across targets, only the basic multilingual plane (BMP) is supported consistently: every BMP Unicode codepoint corresponds to exactly one string character.

It is still possible to work with strings including non-BMP characters on all targets without having to manually decode surrogate pairs by using the Unicode iterators API provided in the standard library.

10.1.3.Encoding

On some targets, the internal representation is UTF-16, which means that non-BMP Unicode codepoints are represented using surrogate pairs. The compile-time define `target.utf16` is set when the target uses UTF-16 internally.

Null-bytes in strings

Some Haxe targets disallow null-bytes (Unicode codepoint 0) in strings. Additionally, some Haxe core APIs assume a null-byte terminates strings. To consistently deal with binary data, including null-bytes, use the `haxe.io.Bytes` API.

Target details

Target	`target.unicode`	`target.utf16`	Internal encoding	Null-byte allowed
Flash	yes	yes	UTF-16	no
JavaScript	yes	yes	UTF-16	yes (except in some old browsers)
ActionScript 3	yes	yes	UTF-16	no
C++	yes	yes	ASCII or UTF-16 (if needed)	yes
Java	yes	yes	UTF-16	yes
JVM	yes	yes	UTF-16	yes
C#	yes	yes	UTF-16	yes
Python	yes	no	Latin-1, UCS-2, or UCS-4 (see PEP 393)	yes
Lua	yes	no	UTF-8	yes
PHP	yes	no	binary	yes
Eval	yes	no	UTF-8	yes
Neko	no	no	binary	yes
HashLink	yes	yes	UTF-16	no

10.2.数据结构

本节内容： 10.2.1: 数组

10.2.2: 向量

10.2.3: 列表

10.2.4: GenericStack

10.2.5: Map

10.2.6: Option

10.2.1.数组

一个数组是一个元素的集合。它有一个类型参数（第3.2节），对应这些元素的类型。数组可以被通过以下三种方式创建：

- 使用它们的构造函数： new Array()
- 使用数组声明语法（第5.5节）： [1, 2, 3]
- 使用数组推导（第6.6节）： [for (i in 0...10) if (i % 2 == 0) i]

数组附带一个 API，覆盖了多数的用例。另外，它们允许读/写 的数组访问（第5.8节）：

```
class Main {
    static public function main() {
        var a = [1, 2, 3];
        trace(a[1]); // 2
        a[1] = 1;
        trace(a[1]); // 1
    }
}
```

由于 Haxe 中的数组访问是没有边界的，即，数组保证不会抛出异常，这需要进一步讨论：

- 如果对一个不存在的索引做出一个读访问，将根据目标语言返回一个值。
- 如果对一个越界的正索引做出一个写访问，则在最后有定义的索引与这个新写入的索引项之间的所有位置插入 null（或者静态目标语言（第2.2节）中基本类型（第2.1节）的默认值（第2.2节））。
- 如果使用一个负索引写访问，结果是未指定的。

数组在它的元素之上定义了一个迭代器（第6.7节）。这个迭代通常由编译器优化后以一个 while 循环（第5.14节）配合数组下标进行：

```
class Main {
    static public function main() {
        var scores = [110, 170, 35];
        var sum = 0;
        for (score in scores) {
            sum += score;
        }
        trace(sum); // 315
    }
}
```

Haxe 生成这个优化后的 JavaScript 输出：

```
Main.main = function() {
    var scores = [110,170,35];
    var sum = 0;
    var _g = 0;
    while(_g < scores.length) {
        var score = scores[_g];
        ++_g;
        sum += score;
    }
    console.log(sum);
};
```

Haxe 不允许混合类型的数组，除非参数类型被限强制为 Dynamic（第2.7节）：

```
class Main {
    static public function main() {
        // Compile Error: Arrays of mixed types are only allowed if the type is
        // forced to Array<Dynamic>
        //var myArray = [10, "Bob", false];

        // Array<Dynamic> with mixed types
        var myExplicitArray:Array<Dynamic> = [10, "Sally", true];
    }
}
```

> 花絮: 动态数组 在Haxe 2中，混合类型的数组声明是被允许的。在Haxe 3中，数组只有显式声明为 Array 才可以为混合类型。

查看数组API了解它的方法的详细内容。

10.2.2.向量

一个向量是一个优化的固定长度的元素集合。很像数组（第10.2.1节），它只有一种类型参数（第3.2节），所有向量的元素必须是特定类型，可以使用一个 for 循环（第5.13节）迭代，并使用数组访问（第2.8.3节）的语法访问。然而，不像数组和立标，向量长度是被创建时指定的，并且之后不能被修改。

```
class Main {
    static function main() {
        var vec = new haxe.ds.Vector(10);

        for (i in 0...vec.length) {
            vec[i] = i;
        }

        trace(vec[0]); // 0
        trace(vec[5]); // 5
        trace(vec[9]); // 9
    }
}
```

haxe.ds.Vector 被实现为一个抽象类型（Abstract）（第2.8节），在给定目标语言的原生数组实现之上，是更快速的固定长度的集合，因为存储它的元素的内存被预分配了。

查看Vector API 来详细了解向量方法。

10.2.3.列表

列表是一个存储的元素的集合。从表面上看，一个列表类似于一个数组。然而，潜在的实现非常不同。这体现在一些功能差异：

- 列表不能被使用方括号索引，即 [0] 形式。
- 列表不能被初始化。
- 没有列表推导。
- 列表在迭代的时候可以自由修改和删除元素。

使用列表的一个简单例子：

```
class Main {
    static public function main() {
        var myList = new List<Int>();
        for (ii in 0...5)
            myList.add(ii);
        trace(myList); //{0, 1, 2, 3, 4}
    }
}
```

查看List API 详细了解列表方法。

10.2.4.GenericStack

一个 GenericStack，就像数组和列表，是一个存储元素的容器。它有一个类型参数（第3.2节），这个堆栈的所有元素必须是特定类型。这里有一个初始化和使用GenericStack的小示例程序。

```
import haxe.ds.GenericStack;

class Main {
    static public function main() {
        var myStack = new GenericStack<Int>();
        for (ii in 0...5)
            myStack.add(ii);
        trace(myStack); //{4, 3, 2, 1, 0}
        trace(myStack.pop()); //4
    }
}
```

> 花絮: FastList 在Haxe 2中， GenericStack类被称为 FastList 。由于它的行为非常接近传统的堆栈，所以在Haxe 3中修改了名字。

GenericStack 中的 Generic 是字面的。它被通过 :generic 元数据归结。根据目标语言，这可以提升静态目标语言的性能。查看 Generic（第3.3节）了解详细内容。

查看GenericStack API 详细了解它的方法。

10.2.5.Map

Map 是一个键值对组成的容器。一个 Map 也通常被称为一个关联数组、字典或者符号表。下面的代码是要给简短的使用Map的示例：

```
class Main {
    static public function main() {
        // Maps are initialized like arrays, but
        // use the map literal syntax with the
        // '=>' operator. Maps can have their
        // key value types defined explicity
        var map1:Map<Int, String> =
        [1 => "one", 2=>"two"];

        // Or they can infer their key value types
        var map2 = [
            "one"=>1,
            "two"=>2,
            "three"=>3
        ];
        $type(map2); // Map<String, Int>

        // Keys must be unique
        // Error: Duplicate Key
        //var map3 = [1=>"dog", 1=>"cat"];

        // Maps values can be accessed using array
        // accessors "[]"
        var map4 = ["M"=>"Monday", "T"=>"Tuesday"];
        trace(map4["M"]); //Monday

        // Maps iterate over their values by
        // default
        var valueSum;
        for (value in map4) {
            trace(value); // Monday \n Tuesday
        }

        // Can iterate over keys by using the
        // keys() method
        for (key in map4.keys()) {
            trace(key); // M \n T
        }

        // Like arrays, a new Map can be made using
        // comprehension
        var map5 = [
            for (key in map4.keys())
                key => "FRIDAY!!"
        ];
        // {M => FRIDAY!!, T => FRIDAY!!}
        trace(map5);
    }
}
```

在后台，Map 是一个抽象类型。在编译时，它被转换为集中特定类型之一，取决于键的类型：

- String： `haxe.ds.StringMap`
- Int : `haxe.ds.IntMap`
- EnumValue： `haxe.ds.EnumValueMap`
- {}： `haxe.ds.ObjectMap`

Map类型在运行时不存在，被上面的对象之一取代。Map使用它的键类型定义数组访问（第2.8.3节）。

查看Map API 详细了解它的方法。

10.2.6.Option

一个 Option 是Haxe标准库中的一个枚举，如下形式定义：

```
enum Option<T> {
    Some(v:T);
    None;
}
```

它可以被使用在各种各样的状况，比如 沟通一个方法是否有一个有效的返回，如果是的话，它返回什么值：

```
import haxe.ds.Option;

class Main {
    static public function main() {
        var result = trySomething();
        switch (result) {
            case None:
                trace("Got None");
            case Some(s):
                trace("Got a value: " +s);
        }
    }

    static function trySomething():Option<String> {
        if (Math.random() > 0.5) {
            return None;
        } else {
            return Some("Success");
        }
    }
}
```

10.3.正则表达式

Haxe 内置支持 正则表达式。它们可以用来验证字符串格式，转换一个字符串，或者从给定文本中提取一些规则数据。

Haxe创建正则表达式有特定的语法。我们可以创建一个正则表达式对象通过输入其到 ~/ 组合和一个单独的 / 符号中：

```
var r = ~/haxe/i;
```

或者，我们可以使用正则语法创建正则表达式：

```
var r = new EReg("haxe", "i");
```

第一个参数是正则表达式模式字符串，第二个是标记字符串（后面查看）。

我们可以使用标准的正则表达式模式，如：

- . 任何字符
- * 重复0或者多次
- + 重复1或者多次
- ? 可选的0或者1次
- [A-Z0-9] 字符范围
- [^\r\n\t] 不在范围内的字符
- (...) 括号匹配字符的分组
- ^ 字符串起始字符（在多行匹配模式中一行的起始字符）
- $ 字符串的尾字符（多行匹配模式下一行的尾字符）
- | "OR"语句

例如，下面的增则表达式匹配有效的 email 地址：

```
~/[A-Z0-9._\%-]+@[A-Z0-9.-]+\.[A-Z][A-Z][A-Z]?/i;
```

注意 正则表达式结尾的 i 是一个标记，作用是启用不区分大小写的匹配。

可能的标记如下：

- i 不区分大小写匹配
- g 全局替换或者分割，查看后面
- m 多行匹配， ^ 和 $ 表示一行的开头和结尾
- s 点号 . 将也匹配新行（Neko，C++，PHP，Flash 和Java 目标）
- u 使用UTF-8匹配（Neko 和 C++目标）

查看EReg API 详细了解它的方法。

10.3.1.匹配

可能正则表达式最常用的一个地方就是检查一个字符串是否匹配特定模式。正则表达式对象的 match 方法可以用来做这些：

```
class Main {
  static function main() {
    var r = ~/world/;
    var str = "hello world";
    // true : 'world' was found in the string
    trace(r.match(str));
    trace(r.match("hello !")); // false
  }
}
```

10.3.2.分组

通过使用分组，特定信息可以被从一个匹配的字符串中提取。如果 match() 返回 true，我们可以使用 matched(X) 方法过的分组，X是正则表达似乎模式定义的分组的号码：

```
class Main {
  static function main() {
    var str = "Nicolas is 26 years old";
    var r =
      ~/([A-Za-z]+) is ([0-9]+) years old/;
    r.match(str);
    trace(r.matched(1)); // "Nicolas"
    trace(r.matched(2)); // "26"
  }
}
```

注意，分组号从1开始，r.matched(0) 总是返回整个匹配的子串。

r.matchedPos() 返回这个子串在原字符串中的位置：

```
class Main {
  static function main() {
    var str = "abcdeeeeefghi";
    var r = ~/e+/;
    r.match(str);
    trace(r.matched(0)); // "eeeee"
    // { pos : 4, len : 5 }
    trace(r.matchedPos());
  }
}
```

另外，r.matchedLeft() 和 r.matchedRight() 可以用来获得匹配的子串左侧或者右侧的子串：

```
class Main {
  static function main() {
    var r = ~/b/;
    r.match("abc");
    trace(r.matchedLeft()); // a
    trace(r.matched(0)); // b
    trace(r.matchedRight()); // c
  }
}
```

10.3.3.替换

一个正则表达式也可以被用于替换字符串的一部分：

```
class Main {
  static function main() {
    var str = "aaabcbcbcbz";
    // g : replace all instances
    var r = ~/b[^c]/g;
    // "aaabcbcbcxx"
    trace(r.replace(str,"xx"));
  }
}
```

在替换中，我们可以使用 $X 来重用一个匹配的组：

```
class Main {
  static function main() {
    var str = "{hello} {0} {again}";
    var r = ~/{([a-z]+)}/g;
    // "*hello* {0} *again*"
    trace(r.replace(str,"*$1*"));
  }
}
```

10.3.4.分割

一个正则表达式也可以用来分割一个字符串到几个子串：

```
class Main {
  static function main() {
    var str = "XaaaYababZbbbW";
    var r = ~/[ab]+/g;
    // ["X","Y","Z","W"]
    trace(r.split(str));
  }
}
```

10.3.5.Map

正则表达式对象的 map 方法可以用于使用一个自定义函数替换匹配的子串。这个函数把一个正则表达式对象作为第一个参数，所以我们可以使用它得到进行匹配的更多信息，并进行条件替换。例如：

```
class Main {
  static function main() {
    var r = ~/(dog|fox)/g;
    var s = "The quick brown fox jumped over the lazy dog.";
    var s2 = r.map(s, function(r) {
        var match = r.matched(0);
        switch (match) {
            case 'dog': return 'fox';
            case 'fox': return 'dog';
            default: throw 'Unknown animal: $match';
        };
    });
    trace(s2); // The quick brown dog jumped over the lazy fox.
  }
```

```
}
```

10.3.6.实现细节

正则表达式的实现：

- 在JavaScript中，运行时使用对象 RegExp 提供实现。
- 在Neko 和C++中，使用 PCRE 库。
- 在Flash、PHP、C#和Java中，使用原生的实现。
- 在Flash 6 /8 中，实现不可用。

10.4.Math

Haxe 包括了一个浮点数数学库，用于一些常见的数学运算。多数函数用于操作和返回浮点数。然而，一个 Int 类型 可以用于接受 Float 的地方，Haxe也会在多数数值运算中转换Int 为 Float（查看数值运算符（第2.1.3）了解更多）。

这里是一些math库使用的例子：

```
class Main {
  static public function main() {
    var x = 1/2;
    var y = 20.2;
    var z = -2;

    trace(Math.abs(z)); //2
    trace(Math.sin(x*Math.PI)); //1
    trace(Math.ceil(y)); //21

    // log is the natural logarithm
    trace(Math.log(Math.exp(5))); //5

    // Output for neko target, may vary
    // depending on platform
    trace(1/0); //inf
    trace(-1/0); //-inf
    trace(Math.sqrt(-1)); //nan
  }
}
```

查看 Math API 了解所有可用的函数。

10.4.1.特殊数值

math 库有一些关于特殊数值的定义：

- NaN（非数字）：当一个不正确的数学操作被执行时返回，例如 Math.sqrt(-1)。
- POSITIVE_INFINITY：如一个整数被 0 除。
- NEGATIVE_INFINITY ：例如一个负数被 0 除 。
- PI ： 3.1415...

10.4.2.数学错误

虽然 neko 可以流畅的处理数学错误，如除数为 0 ，这在所有目标语言都是不正确的。根据不同的目标平台，数学错误可能引发异并引起错误。

10.4.3.整数数学

如果你的目标平台可以利用整数运算，例如整数除法，它可以被包裹到 Std.int() 中 来改进性能。Haxe编译器就可以对整数运算进行优化。一个例子：

```
var intDivision = Std.int(6.2/4.7);
```

10.4.4.扩展

经常看到静态扩展使用 math 库。这个代码展示一个简单的例子：

```
class MathStaticExtension {
  /* Converts an angle in radians to degrees */
  inline public static function toDegrees(radians:Float):Float {
    return radians * 180 / Math.PI;
  }
}
using MathStaticExtension;

class Main {
  public static function main() {
    var ang = 1/2*Math.PI;
    trace(ang.toDegrees()); //90
  }
}
```

10.5.Lambda

> 定义：Lambda Lambda 是一个函数式语言概念，在Haxe中它允许你应用一个函数到一个列表或者迭代（第6.7节）。Lambda 类是一个函数方法的集合，以使Haxe使用函数风格编程。

通过 using Lambda（查看静态扩展（第6.3节））它被完美的使用，然后作为一个 Iterator 类型的扩展。

在静态目标平台，使用 Iterator 结构可能慢于直接对已知类型执行操作，比如 Array 和 List 。

Lambda 函数

Lambda 类允许我们一次操作一整个迭代。这通常是更好的循环方式，因为它不太容易出错，并且更易读取。方便起见，Array 和 List 类包含一些频繁使用的Lambda 类中的方法。

查看一个例子。exists 函数被指定为：

```
static function exists<A>( it : Iterable<A>, f : A -> Bool ) : Bool
```

多数Lambda 函数以同样的方式调用。所有Lambda 函数的第一个参数都是要操作目标的 Iterator 。有许多也把一个函数作为一个参数。

- **Lambda.array**，**Lambda.list** 转换Iterator 为 Array 或 List 。它总是返回一个新的实例。
- **Lambda.count** 计算元素的数量。如果可迭代的是一个 Array 或者 List，则使用它们更快速的 length 属性。
- **Lambda.empty** 确定是否可迭代对象为空。对于所有可迭代对象最好使用这个函数；也快于对比length属性（或者 Lambda.count 的结果）是否为0。
- **Lambda.has** 确定是否特定元素存在于可迭代对象中。
- **Lambda.exists** 确定是否条件是可以被一个元素满足的。
- **Lambda.indexOf** 找出某个特定元素的索引。
- **Lambda.find** 找到给定搜索函数的第一个元素。
- **Lambda.foreach** 确定是否每个元素满足一个条件。
- **Lambda.iter** 对每个元素调用一个函数。
- **Lambda.concat** 合并两个可迭代对象，返回一个新的 List 。
- **Lambda.filter** 找到满足一个条件的元素，返回一个新的 List 。
- **Lambda.map**， **Lambda.mapi** 应用一个转换到每个元素，返回一个新的 List 。
- **Lambda.fold** 功能性折叠，也被称为缩小、积聚、压缩或者注入。

这个例子演示了Lambda的 filter 和 map 函数对一组字符串的操作：

```
using Lambda;
class Main {
    static function main() {
        var words = ['car', 'boat', 'cat', 'frog'];

    var isThreeLetters = function(word) return word.length == 3;
    var capitalize = function(word) return word.toUpperCase();

    // Three letter words and capitalized.
    trace(words.filter(isThreeLetters).map(capitalize)); // [CAR,CAT]
    }
}
```

这个例子演示了 Lambda的 count，has ，foreach 和 fold 函数操作一组整数：

```
using Lambda;
class Main {
    static function main() {
        var numbers = [1, 3, 5, 6, 7, 8];

    trace(numbers.count()); // 6
    trace(numbers.has(4)); // false

        // test if all numbers are greater/smaller than 20
    trace(numbers.foreach(function(v) return v < 20)); // true
        trace(numbers.foreach(function(v) return v > 20)); // false

        // sum all the numbers
    var sum = function(num, total) return total += num;
    trace(numbers.fold(sum, 0)); // 30
    }
}
```

查看 Lambda API 了解所有可用的函数。

10.6.模板

Haxe 附带了一个标准的模板系统，使用被一个轻量的 haxe.Template 类解释的简单的语法。

模板是一个字符串或者一个文件，用来产生任何种类的取决于输入内容的字符串输出。这是一个小的模板例子：

```
class Main {
  static function main() {
    var sample = "My name is <strong>::name::</strong>, <em>::age::</em> years old";
    var user = {name:"Mark", age:30};
    var template = new haxe.Template(sample);
    var output = template.execute(user);
    trace(output);
  }
}
```

控制台会输出： `The console will trace My name is Mark, 30 years old.`

表达式

- `::name::` 变量名
- `::expr.field::` 字段访问
- `::(expr)::` 表达式 expr 被评估
- `::(e1 op e2)::` 对 e1 和 e2 应用操作符 op
- `::(135)::` 整数 135 。Float类型常量是不允许的

条件

可以使用 `::if flag1::` 测试条件。可选地，条件可以使用 `::elseif flag2::` 或者 `::else::` 。关闭条件使用 `::end::` 。

```
::if isValid:: valid ::else:: invalid ::end::
```

运算符可以被使用，但是不处理运算符优先级。因此需要需要把每个操作放入括号 `()` 。目前，如下的运算符被允许： `+` ， `-` ， `*` ， `/` ， `>` ， `<` ， `>=` ， `<=` ， `==` ， `!=` ， `&&` ，和 `||` 。

例如， `::((1 + 3) == (2 + 2))::` 会显示 true 。

```
::if (points == 10):: Great! ::end::
```

要比较一个字符串，使用双引号 " 到模板中。

```
::if (name == "Mark"):: Hi Mark ::end::
```

迭代

通过使用 `::foreach::` 迭代一个结构。结束循环使用 `::end::` 。

```
<table>
  <tr>
    <th>Name</th>
    <th>Age</th>
  </tr>
  ::foreach users::
    <tr>
      <td>::name::</td>
      <td>::age::</td>
    </tr>
  ::end::
</table>
```

子模板

要在一个模板中包括另一个模板，传递子模板结果字符串作为一个参数。

```
var users = [{name:"Mark", age:30}, {name:"John", age:45}];

var userTemplate = new haxe.Template("::foreach users:: ::name::(::age::) ::end::");
var userOutput = userTemplate.execute({users: users});

var template = new haxe.Template("The users are ::users::");
var output = template.execute({users: userOutput});
trace(output);
```

控制台会输出： `The users are Mark(30) John(45).`

模板宏

当部分模板被渲染时要调用自定义的函数，提供一个 `宏(macros)` 对象到 Template.execute 的参数即可。键会作为模板变量名，值引用一个应该返回 `String` 的回调函数。这个宏函数的第一个参数总是一个 `resolve()` 方法，后跟指定的参数。 resolve 函数可以被调用来从模板上下文中取回值。如果 `macros` 没有这个字段，结果将是未指定的。

下面的例子传递它本身作为宏函数上下文，并从模板执行 `display`：

```
class Main {
  static function main() {
    new Main();
  }

  public function new() {
    var user = {name:"Mark", distance:3500};
    var sample = "The results: $$display(::user::,::time::)";
    var template = new haxe.Template(sample);
    var output = template.execute({user:user, time: 15}, this);
    trace(output);
  }

  function display(resolve:String->Dynamic, user:User, time:Int) {
    return user.name + " ran " + (user.distance/1000) + " kilometers in " + time + " minutes";
  }
}
typedef User = {name:String, distance:Int}
```

控制台会输出如下结果：`Mark ran 3.5 kilometers in 15 minutes.`

全局

使用 Template.globals 对象来存储可以被贯穿整个 `haxe.Template` 实例中应用的值。它的优先级低于 `Template.execute` 的上下文参数。

利用资源

要从代码中分离内容，考虑使用资源嵌入系统（第8.4节）。存放模板内容到一个新的文档，叫做 `sample.mtt`，添加 `-resource sample.mtt@my_sample` 到编译器参数，使用 `haxe.Resource.getString` 来检索内容。

```
class Main {
  static function main() {
    var sample = haxe.Resource.getString("my_sample");
    var user = {name:"Mark", age:30};
    var template = new haxe.Template(sample);
    var output = template.execute(user);
    trace(output);
  }
}
```

当在服务端运行模板系统，你可以简单的使用 `neko.Lib.print` 或者 `php.Lib.print`，而不是使用 trace 来显示 HTML 模板给用户。

查看 Template API 详细了解它的方法。

10.7.反射

Haxe 支持运行时的类型和字段反射。必须注意，因为运行时表示不同目标平台之间的一般差异。为了正确的使用反射，需要理解哪种类别的操作是被支持的，哪些不是。鉴于反射的动态特性，这不能总是被在编译时确定。

反射API由两个类组成：

- Reflect： 一个轻量级的API，很好的使用于匿名结构，但是对类的有限支持。
- Type： 适用于类和枚举的更加健壮的API 。

可用的方法在 Reflect 和 Type API中详细介绍。

Reflection 是一个强大的工具，但是重要的是，理解它为什么也可能引起问题。举个例子，一些函数接受一个 String 参数，并试图解析它为一个类型或者字段。这很容易带来类型错误：

```
class Main {
  static function main() {
    trace(Type.resolveClass("Mian")); // null
  }
}
```

然而，即使没有类型错误，也很容易遇到意料之外的行为：

```
class Main {
  static function main() {
    // null
    trace(Type.resolveClass("haxe.Template"));
  }
}
```

这里的问题是，编译器从来没有真正的"看到“ haxe.Template 类型，所以它不会编译输出。此外，即使它看到过这个类型，仍然可能从有问题发生，在无用代码消除（第8.2节）消除的仅通过反射使用的类型或字段。

另一组问题来自于一个事实，一些反射函数有意的接受 Dyanmic（第2.7节） 类型的参数，也就是说，编译器不能检查在参数中传递的是不是正确的。如下的例子演示了当使用 callMethod 时一个常见的错误：

```
class Main {
  static function main() {
    // wrong
    //Reflect.callMethod(Main, "f", []);
    // right
    Reflect.callMethod(Main,
      Reflect.field(Main, "f"), []);
  }

  static function f() {
    trace('Called');
  }
}
```

注释掉的调用可以被编译器接受，因为它分配字符串 “f” 到被指定为 Dynamic 类型的函数参数 func 。

当使用反射时一个好的建议是，包装它到被原本类型安全的代码调用的一个应用或者API的一些函数中。看一下这个例子：

```
typedef MyStructure = {
  name: String,
  score: Int
}

class Main {
  static function main() {
    var data = reflective();
    // At this point data is nicely typed as MyStructure
  }

  static function reflective():MyStructure {
    // Work with reflection here to get some values we want to return.
    return {
      name: "Reflection",
      score: 0
    }
  }
}
```

尽管方法 reflective 能和反射一起使用（对于这个事件，而且是Dynamic），但它返回值是一个类型化的结构，可以使调用者使用类型安全的方式。

10.8.序列化

许多运行时值可以被序列化或反序列化，使用 haxe.Serializer 和 haxe.Unserializer 类。都支持两种用法：

创建一个实例，并不断的调用 serialize / unserialize 方法来处理多个值。 调用它们的静态 run 方法来 序列化/反序列化 一个单独的值。

下面的例子演示了第一种用法：

```
import haxe.Serializer;
import haxe.Unserializer;

class Main {
  static function main() {
    var serializer = new Serializer();
    serializer.serialize("foo");
    serializer.serialize(12);
    var s = serializer.toString();
    trace(s); // y3:fooi12

    var unserializer = new Unserializer(s);
    trace(unserializer.unserialize()); // foo
    trace(unserializer.unserialize()); // 12
  }
}
```

序列化的结果（这里存储在局部变量 s）是一个 String，并且可以被任意甚至远程地传递。它的格式以 序列化格式（第10.8.1节）描述。

支持的值

- null
- Bool， Int 和 Float （包括无穷值和 NaN）
- String
- Date
- haxe.io.Bytes （编码为 base64）
- Array 和 List
- haxe.ds.StringMap， haxe.ds.IntMap 和 haxe.ds.ObjectMap
- 匿名结构
- Haxe 类实例（并不是原生的）
- 枚举实例

序列化配置

序列化可以以两种方式配置。对于一个静态变量，可以被设置来改变所有 haxe.Serializer 实例，一个成员变量可以被设置来影响一个特定实例：

- USE_CACHE，userCache ： 如果为 true，重复的结构或类/枚举实例被参照序列化。这可以避免递归数据的无限循环更长的序列化时间。默认，对象缓存是禁用的；然而字符串总是被缓存。
- USE_ENUM_INDEX，useEnumIndex ： 如果为 true ，枚举构造函数被它们的索引序列化而不是它们的名

字。这可以使结果字符串更短，但是如果 枚举构造函数在反序列化之前被插入到这个类型，将会打断。这个行为默认是禁止的。

反序列化行为

如果序列化结果被存储，之后使用于反序列化，必须注意当使用类和枚举实例时要保持兼容性。之后重要的是准确理解反序列化如何实现的。

- 做反序列化的地方类型必须在运行时可以获得的。如果无用代码消除被激活，只是通过序列化使用的类型可能会被删除。
- 每个 Unserializer 都有一个成员变量 resolver ，用于通过名字解析类和枚举。Unserializer 一经创建，它被设置为 Unserializer.DEFAULT_RESOLVER 。它和实例成员都可以被设置为一个定制 分析器。
- 类使用 resolver.resolveClass(name) 通过名字解析。实例然后被使用 Type.createEmptyInstance 创建，这意味着 这个类构造函数没有被调用。最终，实例字段根据序列化的值被设置。
- 枚举使用 resolver.resolveEnum(name)通过名字及诶系。枚举实例然后被使用 Type.createEnum创建，如果可用，则使用序列化的参数值。如果构造函数参数由于序列化被改变，结果是未指定的。

自定义的（反）序列化

如果一个类定义了成员方法 hxSerialize，这个方法被序列化器调用，允许对类的自定义序列化。同样，如果一个类定义了成员方法 hxUnserialize，它被反序列化器调用：

```
import haxe.Serializer;
import haxe.Unserializer;

class Main {

  var x:Int;
  var y:Int;

  static function main() {
    var s = Serializer.run(new Main(1, 2));
    var c:Main = Unserializer.run(s);
    trace(c.x); // 1
    trace(c.y); // -1
  }

  function new(x, y) {
    this.x = x;
    this.y = y;
  }

  @:keep
  function hxSerialize(s:Serializer) {
    s.serialize(x);
  }

  @:keep
  function hxUnserialize(u:Unserializer) {
    x = u.unserialize();
    y = -1;
  }
}
```

在这个例子中，我们决定要忽略成员变量 y的值，并且不序列化它。相反，我们在 hxUnserialize中默认它为 -1 。两个方法都使用 @:keep 元数据注解，以防止无用代码消除删除它们，因为它们在代码中从未恰当的引用。

查看 Serializer 和 Unserializer API 文档了解详细内容。

10.8.1.格式化序列化

每个支持的值被转换为一个不同前缀字符、后跟需要的数据。

- null: n
- Int: 0为z，或者 i 后跟整数显示（例如 i456）
- Float:
  - NaN ： k
  - 负无穷 ： m
  - 正无穷 ： p
  - 有限的浮点数 ： d 后跟浮点数显示（如d1.45e-8）
- Bool ：true 为 t，false 为 f
- String: y 后跟url编码的字符串长度，然后是冒号 : 和url编码（如 y10:hi%20there,“hi there”）
- 名称-值 对: 一个序列化的字符串表示名称后跟序列化的值
- 结构: o 后跟 名称-值对的列表，以g 终止？（例如，oy1:xi2y1:kng，{x:2, k:null}）
- List: l 后跟序列化的项的列表，后跟h（例如，lnnh ，两个null值的 List）
- Array: a 后跟序列化的项的列表，后跟h。对于多个连续的 null 值， 使用u 后跟null的数量（例如 ai1i2u4i7ni9h ， [1, 2, null, null, null, null, 7， null, 9]）
- Date: v 后跟 date 自己（例如 v2010-01-01 12：45：10）
- haxe.ds.StringMap: b 后跟 名-值对，后跟 h（例如，by1:xi2y1:knh，{"x"=> 2, "k"=> null}）
- haxe.ds.IntMap: q 后跟键值对，后跟h。每个键表示为 :（例如，q:4n:5i45:6i7h ，{4=>null, 5=> 45, 6=>7}）
- haxe.ds.ObjectMap: M 后跟序列化的值对表示键跟值，后跟h
- haxe.io.Bytes: s 后跟 base64编码字节长度，然后冒号 : ，和A-Za-z0-9% 表示的字节码（例如，s3:AAA ，等于0的2字节，s10:SGVsbG8gIQ ，haxe.io.Bytes.ofString("Hello !")）
- exception: x 后跟异常值
- 类实例: c 后跟序列化的类名，后跟字段的名-值对，后跟g（例如，cy5:Pointy1:xzy1:yzg ，new Point(0, 0)）（有两个整数字段 x 和 y）
- 枚举实例 (通过名称): w 后跟序列化的枚举名称，后跟序列化的构造函数名，后跟冒号 : ，后跟参数的数量，后跟参数的值（例如，wy3:Fooy1:A:0 ，Foo.A （无参数），wy3:Fooy1:B:2i4n ，Foo.B(4, null)）
- 枚举实例(通过索引): j 后跟序列化的枚举名称，后跟冒号 : ，后跟构造函数的索引（从0开始），后跟冒号 :，后跟参数的数量，后跟参数的值（例如，jy3:Foo:0:0 ， Foo.A （无参数）， jy3:Foo:1:2i4n ，Foo.B(4, null)）
- 缓存引用:
  - String: R 后跟在字符串缓存中相应的索引（例如，R456）
  - class, enum 或 structure ： r 后跟对象缓存中的相应索引（如，r42）
- 自定义: C 后跟类名，后跟自定义序列化数据，后跟 g

缓存的元素和枚举构造函数从0开始索引。

10.9.Xml

Haxe 通过 haxe.XML 类内置提供对使用XML数据的支持。

10.9.1.开始使用Xml

创建一个根元素

Xml的一个根元素可以使用 Xml.createElement 方法创建。

```
var root = Xml.createElement('root');
trace(root); // <root />
```

一个根节点元素也可以通过解析一个字符串包含的 XML 数据来创建。

```
var root = Xml.parse('<root />').firstElement();
trace(root); // <root />
```

创建子元素

可以使用 addChild 方法添加子元素到根。

```
var child:Xml = Xml.createElement('child');
root.addChild(child);
trace(root); // <root><child/></root>
```

可以使用 set() 方法为元素添加属性。

```
child.set('name', 'John');
trace(root); // <root><child name="John"/></root>
```

访问元素和值

这段代码解析一个 XML 字符串为一个 对象结构 Xml ， 然后访问对象的属性。

```
var xmlString = '<hello name="world!">Haxe is great!</hello>';
var xml:Xml = Xml.parse(xmlString).firstElement();

trace(xml.nodeName); // hello
trace(xml.get('name')); // world!
trace(xml.firstChild().nodeValue); // Haxe is great!
```

firstChild 和firstElement 之间的不同是，第二个函数返回的第一个子元素为Xml.Element 类型。

迭代 Xml 元素

我也也可以用其它方法迭代每个子节点或元素。

```
for (child in xml) {
  // iterate on all children.
}
for (elt in xml.elements()) {
  // iterate on all elements.
}
for (user in xml.elementsNamed("user")) {
  // iterate on all elements with a nodeName "user".
}
for (att in xml.attributes()) {
  // iterator on all attributes.
}
```

查看 Xml API 文档详细了解它的方法。

10.9.2.解析Xml

静态方法 Xml.parse 可以用来解析 XML 数据 并从其中获得一个 Haxe 值。

```
var xml = Xml.parse('<root>Haxe is great!</root>').firstElement();
trace(xml.firstChild().nodeValue);
```

10.9.3.编码Xml

xml.toString() 方法可以用来获取字符串的表示。

```
var xml = Xml.createElement('root');
xml.addChild(Xml.createElement('child1'));
xml.addChild(Xml.createElement('child2'));

trace(xml.toString()); // <root><child1/><child2/></root>
```

10.10.Json

Haxe 通过 haxe.Json 类内置支持（反）序列化 JSON 数据。

10.10.1.解析JSON

使用 haxe.Json.parse 静态方法 来解析 JSON 数据，并从中获取一个 Haxe 值：

```
class Main {
  static function main() {
    var s = '{"rating": 5}';
    var o = haxe.Json.parse(s);
    trace(o); // { rating: 5 }
  }
}
```

注意，haxe.Json.parse 返回的对象的类型是 Dynamic ，所以如果我们数据的结构如果是已知的，我们可能想要使用匿名结构（第2.5节）指定类型。这个方式我们提供对访问我们的数据的编译时检查，并生成几乎最优的代码，因为编译器了解结构中的类型：

```
typedef MyData = {
  var name:String;
  var tags:Array<String>;
}

class Main {
  static function main() {
    var s = '{
      "name": "Haxe",
      "tags": ["awesome"]
    }';
    var o:MyData = haxe.Json.parse(s);
    trace(o.name); // Haxe (a string)
    // awesome (a string in an array)
    trace(o.tags[0]);
  }
}
```

10.10.2.编码JSON

使用 haxe.Json.stringify 静态方法来编码一个 Haxe 值到 JSON 字符串：

```
class Main {
  static function main() {
    var o = {rating: 5};
    var s = haxe.Json.stringify(o);
    trace(s); // {"rating":5}
  }
}
```

10.10.3.实现细节

haxe.Json API 自动使用原生的实现在它可用的目标平台，即 JavaScript ， Flash 和PHP ， 并对其它的目标平台提供它们自己的实现。

Haxe自身实现的用法可以强制使用 -D haxeJSON 编译器标记开启。这也提供通过索引、字符串类型键的map（第10.2.5节）和类实例序列化枚举（第2.4节）。

旧版的浏览器（例如IE7）可能没有原生的 JSON 实现。如果需要支持它们，我们可以在HTML页面包含一个在互联网上的可用的JSON实现。另外 ， 一个 -D old_browser 编译器标记可以使 haxe.Json尝试使用原生 JSON，如果不被支持，则回退到它自己的实现。

10.11.Input/Output

官方文档暂缺。

10.12.Sys

The majority of Haxe targets are so-called "sys" targets. This means the targets have access to system APIs such as the filesystem, networking, threads, and more. The only non-sys targets supported by Haxe are Flash, JavaScript, and ActionScript 3, although JavaScript can support sys APIs when running under Node.js runtime.

Related content

- See the sys package on the API documentation for more details on its classes.

10.12.1.Threading

since Haxe 4.0.0

A unified threading API is available on some sys targets. The compile-time define `target.threaded` is set when the API is available. The API allows very simple creation of threads from functions:

```
class Main {
  public static function main():Void {
    #if (target.threaded)
    sys.thread.Thread.create(() -> {
      while (true) {
        trace("other thread");
        Sys.sleep(1);
      }
    });
    Sys.sleep(3);
    #end
  }
}
```

All spawned threads are treated as daemon threads, meaning that the main thread will not wait for their completion.

Due to threads having access to a shared memory space with all the Haxe variables and objects, it is possible to run into issues due to deadlocks and race conditions. The standard library provides some core synchronization constructs in the `sys.thread` package.

10.13.远程处理

Haxe 远程处理是一个在不同平台之间通讯的方式。通过Haxe的远程处理，应用可以透明的传送数据，发送数据，和在服务端/客户端之间调用方法。

在API 文档中查看 remoting 包了解它的类的更多细节。

10.13.1.远程连接

为了使用 remoting ，必须有一个连接建立。有两种Haxe 远程连接：

- haxe.remoting.Connection 是用于同步连接，调用一个方法时结果可以直接获得。
- haxe.remoting.AsyncConnection 用于异步连接，结果是在执行过程中之后发生的事件。

启动一个连接

有一些目标平台特定的构造函数用于不同的目的，可以用来设置一个连接：

- 所有目标平台
  - `HttpAsyncConnection.urlConnect(url:String)` ： 返回一个到指定URL的异步连接，可以连接到一个Haxe服务端应用。
- Flash
  - `ExternalConnection.jsConnect(name:STRING, ctx:Context)` ： 允许一个到本地JavaScript Haxe 代码的连接。JS Haxe代码必须被包含 ExternalConnection 类进行编译。这只适用于Flash Player 8 和更高版本。
  - `AMFConnection.urlConnect(url:String)` 和 `AMFConnection.connect(cnx: NetConnection)` ： 允许到一个AMF远程服务器（如Flash 媒体服务器或者 AMFPHP）的连接。
  - `SocketConnection.create(sock:flash.XMLSocket)` ： 允许在XMLSocket上进行远程通讯。
  - `LocalConnection.connect(name:String)` ： 允许在Flash LocalConnection上进行远程通讯。
- JavaScript
  - ExternalConnection.flashConnect(name:String, obj:String, ctx:Context) 允许连接到一个指定的Flash对象。Haxe Flash内容必须被加载，并且它必须包含 haxe.remoting.Connection 类。只用于Flash8 或更高版本。
- Neko
  - HttpConnection.urlConnect(url:String) 像异步版本一样使用，但是在同步模式下工作。
  - SocketConnection.create(...) 允许实时和一个使用一个XML Socket来连接服务的Flash客户端通讯。

远程上下文

在平台之间通讯之前，一个远程处理的上下文必须被定义。这是一个可以被客户端代码的连接上调用的共享的API。

这个服务代码示例创建和共享一个API ：

```
class Server {
  function new() { }
  function foo(x, y) { return x + y; }
```

```
  static function main() {
    var ctx = new haxe.remoting.Context();
    ctx.addObject("Server", new Server());

    if(haxe.remoting.HttpConnection.handleRequest(ctx))
    {
      return;
    }

    // handle normal request
    trace("This is a remoting server !");
  }
}
```

使用连接

连接使用起来非常方便。一旦连接被获得，使用经典的点语法来屏幕一个路径，然后使用 call() 调用远程上下文中的方法，并获得结果。异步连接接受一个附加的函数参数，在结果获得之后被调用。

客户端代码示例连接到服务端远程上下文并调用这个API上的一个函数 foo()。

```
class Client {
  static function main() {
    var cnx = haxe.remoting.HttpAsyncConnection.urlConnect("http://localhost/");
    cnx.setErrorHandler( function(err) trace('Error: $err'); } );
    cnx.Server.foo.call([1,2], function(data) trace('Result: $data'););
  }
}
```

要使它用于Neko目标平台，设置一个Neko Web 服务器，指向Client中的url到 “http://localhost:2000/remoting.n” ，并使用 -main Server -neko remoting.n 编译 Server 。

错误处理

- 当在异步调用中发生一个错误，就像上面例子中看到的，错误处理程序将被调用。
- 当在同步调用中发生一个错误，在访客端一个异常被抛出，就像我们调用一个本地方法。

数据序列化

Haxe 远程处理可以发送多种不同种类的数据。查看 序列化（第10.8节）。

查看API文档中的 remoting 包 详细了解它的类。

10.13.2.实现细节

JavaScript 安全细节

包含js客户端的html页面必须在跟服务器运行的同样的域名下提供服务。同源策略限制一个源上的一个文档或脚本如何和另一个源上的资源交互。同源策略作为一种手段来防止一些跨站点请求伪造攻击。

要跨域使用remoting，需要通过在 .htaccess 中定义头 X-Haxe-Remotinig 来启用CORS（cross-origin资源共享）。

```
# Enable CORS
Header set Access-Control-Allow-Origin "*"
Header set Access-Control-Allow-Methods: "GET,POST,OPTIONS,DELETE,PUT"
Header set Access-Control-Allow-Headers: X-Haxe-Remoting
```

查看同源策略了解更多这个话题的信息。

也要注意，这意味着页面不能被从文件系统直接提供服务：

```
"file:///C:/example/path/index.html"
```

Flash安全细节

当 Flash 访问一个不同域名的服务器，设置服务器上一个 crossdomain.xml 文件，使用 X-Haxe 头。

```
<cross-domain-policy>
  <allow-access-from domain="*"/> <!-- or the appropriate domains -->
  <allow-http-request-headers-from domain="*" headers="X-Haxe*"/>
</cross-domain-policy>
```

不确保参数类型

关于在一个方法被远程调用时参数类型会被遵守，没有任何种类的保证。也就是说即使函数foo的参数类型化为 Int，客户端仍然可以在调用方法时使用字符串作为参数。这可以导致一些情况下的安全问题。当不确定时，在函数被调用使用 Std.is 方法检查参数类型。

10.14.单元测试

Haxe标准库在 haxe.unit 包中提供基础的单元测试类。

创建新的测试用例

首先，创建一个新的类，继承自 haxe.unit.TestCase ，并添加自己的测试方法。每个测试方法名字都是由 “test”开始。

```
class MyTestCase extends haxe.unit.TestCase {
  public function testBasic() {
    assertEquals("A", "A");
  }
}
```

运行单元测试

要运行测试， haxe.unit.TestRunner 的一个实例必须被创建。使用 add 方法添加 TestCase ，并调用 run 开始测试。

```
class Main {
  static function main() {
    var r = new haxe.unit.TestRunner();
    r.add(new MyTestCase());
    // add other TestCases here

    // finally, run the tests
    r.run();
  }
}
```

测试的结果大致如下：

```
Class: MyTestCase
.
OK 1 tests, 0 failed, 1 success
```

测试函数

haxe.unit.TestCase 类有三个测试函数。

- `assertEquals(a,b)` 如果a和b相等则成功，a是测试的值，b是期望的值。
- `assertTrue(a)` 如果a为 true 则成功。
- `assertFalse(a)` 如果a 为 false 则成功。

安装和拆卸

要在测试之前或之后运行代码，可以重载TestCase 中的 setup 和 tearDown 。

- setup 在每个测试运行之前调用。
- tearDown 在每个测试运行之后调用。

```
class MyTestCase extends haxe.unit.TestCase {
  var value:String;

  override public function setup() {
    value = "foo";
  }

  public function testSetup() {
    assertEquals("foo", value);
  }
}
```

比较复杂对象

使用复杂对象，可能难于生成预期的值来比较实际值。assertEquals 不做深度的对比也可能是一个问题。解决这些问题的一个方法是使用一个字符串作为预期的值，并对比它和使用 Std.string 转换为字符串的实际值 。下面是使用数组的一个简单示例：

```
public function testArray() {
  var actual = [1,2,3];
  assertEquals("[1, 2, 3]", Std.string(actual));
}
```

在API文档中查看 haxe.unit 包了解更多细节。